

AFIT/DS/ENG/96-02

A Generic Intelligent Architecture for
Computer-Aided Training of Procedural Knowledge

DISSERTATION
Freeman Alexander Kilpatrick, Jr
Captain, USAF

AFIT/DS/ENG/96-02

19970317 025

DTIC QUALITY INSPECTED 3

Approved for public release; distribution unlimited

AFIT/DS/ENG/96-02

A Generic Intelligent Architecture for
Computer-Aided Training of Procedural Knowledge

DISSERTATION

Presented to the Faculty of the Graduate School of Engineering
of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

Freeman Alexander Kilpatrick, Jr, B.S.E.E., M.S.S.M., M.S.C.E.

Captain, USAF

March 6, 1996

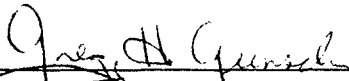
Approved for public release; distribution unlimited

A Generic Intelligent Architecture for
Computer-Aided Training of Procedural Knowledge

Freeman A. Kilpatrick Jr., B.S.E.E., M.S.S.M, M.S.C.E.


Captain, USAF

Approved:



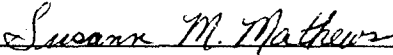
Gregg H. Gunsch, Major, USAF
Chairman, Advisory Committee

6 March 96
Date



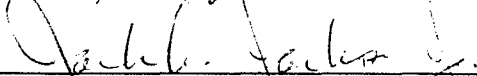
Eugene Santos Jr.
Member, Advisory Committee

23 Feb 96
Date



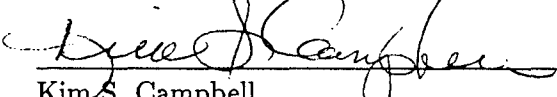
Susann M. Mathews
Member, Advisory Committee

23 Feb 96
Date




Jack A. Jackson Jr., Lt Col, USAF
Member, Advisory Committee

21 Mar 96
Date



Kim S. Campbell
Member, Advisory Committee

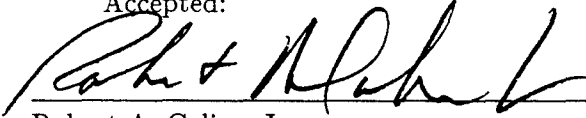
21 Feb 96
Date



Kenneth W. Bauer, Lt Col, USAF
Dean's Representative

5 Mar 96
Date

Accepted:



Robert A. Calico, Jr.
Dean, Graduate School of Engineering

Preface

This dissertation is dedicated to my sons, Adam and Sanders, both of whom were born during my Ph.D. program. Watching the miracle of their developing intelligence reminded me daily that machine intelligence is but a pale shadow of the real thing

To pursue a Ph.D. requires at least a small degree of insanity. Fortunately, there are many people that a PhD candidate works with that help keep this small measure of insanity from becoming a full fledged psychosis. First of all, I would like to express my sincere appreciation to my advisor, Maj Gregg Gunsch, whom I have worked with for almost five years. Our debates, both technical and philosophical were greatly appreciated and will be sorely missed. Also, his ability to instantly see the holes in new ideas and fields both infuriated and amazed me. I would also like to thank my other committee members: Dr. Gene Santos for a (mostly failed) attempt to keep me from becoming too scruffy, and for reminiscing about the classic video games of our youth; Dr. Kim Campbell for her humor and relentless style suggestions; Dr. Susann Mathews for her educational perspective and general sanity-preserving advice. Additionally, I would like to thank my sponsor, LtCol Nancy Crowley for her support of computer resources and TDY funding.

Although the committee is a major driving factor for the successful completion of a Ph.D., there are many "unofficial" contributors as well. I would like to express my appreciation to Doug Dyer, my "upperclassman," for his encouragement and pragmatic advice throughout the entire program. Also, Dennis Montera and Steve Forsythe provided some welcome lunchtime diversions to give my mind a break from the rigors of research. Bill Wood and Pete Collins provided some mostly bad exchanges of music in attempt to broaden my sense of culture, and a daily trek to the vital AFIT bookstore for caffeine supplements. Also, my fellow M.S. extendees: Frank Young, Bruce Anderson and Larry Merkle were a welcome source of commiseration.

Finally, and most importantly I have to express my deepest appreciation to my family. My parents, Freeman and Sara Kilpatrick have given me unfailing support, love and encouragement through twenty-five straight years of school. My wonderful wife, Donna, and my sons Adam and Sanders have contributed to my success more than they can ever

know. They were my bedrock of reality; their love and support allowed me to keep a healthy perspective, and made the completion of the degree meaningful. I couldn't have done it without them.

Freeman Alexander Kilpatrick, Jr

Table of Contents

	Page
Preface	iii
List of Figures	x
Abstract	xii
I. Introduction	1
1.1 Background	1
1.2 Overview	2
1.2.1 The Problem	2
1.2.2 Motivation	3
1.2.3 Approach	3
1.3 Scope	4
1.4 Objectives	5
1.5 Research Structure	6
II. Background	7
2.1 History	7
2.2 Intelligent Tutoring Systems	9
2.2.1 The Expert Module	9
2.2.2 The Student Model	9
2.2.3 The Tutor Module	11
2.3 Related Concepts	11
2.3.1 Shallow vs. Deep Knowledge	11
2.3.2 Simulations	11
2.3.3 Machine Learning	14
2.4 Educational Theory	14

	Page
2.4.1 Learning Theory	14
2.4.2 Operator Decision-Making Theory	16
2.4.3 Training Theory	19
2.4.4 Practice Theory	21
2.5 Related Research	22
2.5.1 ACQUIRE-ITS	22
2.5.2 ITSIE – Intelligent Training Systems in Industrial Environments	23
2.5.3 NASA’s Intelligent Computer-Aided Training Authoring Environment	23
2.5.4 Intelligent Simulation Training System (ISTS)	24
III. Architecture	25
3.1 Knowledge Requirements	26
3.1.1 Training Simulation	28
3.1.2 Shallow Domain Knowledge	29
3.1.3 Summary	31
3.2 Architecture	31
3.2.1 Overview	32
3.2.2 The Simulation	33
3.2.3 The Domain Expert Knowledge Base	36
3.2.4 Knowledge Acquisition Module	36
3.2.5 Control Module	37
3.2.6 Tutor Module	37
3.2.7 Student Model	38
3.2.8 Expert Module	38
3.2.9 Scenario Library	38
3.3 Summary	38

	Page
IV. Knowledge Acquisition	40
4.1 Accessing Internal Representations	40
4.1.1 Knowledge Base Internals	40
4.1.2 Simulation Internals	43
4.1.3 Summary	44
4.2 Machine Learning	44
4.2.1 An Illustration of The Problem	44
4.2.2 Induction	47
4.2.3 Summary	56
4.3 Scenario Exploration	56
4.3.1 The Problem	56
4.4 Action-Based Exploration	57
4.4.1 Simulation Graphs	59
4.4.2 The Exploration Process	61
4.4.3 Summary	64
4.5 Consistency Checking	64
4.6 Curriculum Extraction	66
4.7 Summary	67
V. Implementation Issues	68
5.1 Introduction	68
5.2 Prototype Architecture	70
5.2.1 Knowledge Base	71
5.2.2 Simulation	72
5.2.3 SCIUS Prototype Framework	73
5.2.4 Summary	74
5.3 Simulation Issues	74
5.3.1 Interface	74

	Page
5.3.2 Exploration Combinatorics	76
5.4 Knowledge Base Issues	81
5.4.1 Interface	81
5.4.2 Timing	82
5.5 Induction Issues	83
5.5.1 Number of Examples	84
5.5.2 Induction Behavior	85
5.5.3 Bias	89
5.6 Limitations of the Model	89
5.6.1 Domain-Specific Knowledge Requirements	89
5.6.2 Shallow Knowledge	90
5.6.3 Knowledge Base Paradigms	91
5.7 Summary	92
VI. Conclusions & Recommendations	93
6.1 Summary	93
6.2 Objectives	94
6.3 Contributions	96
6.3.1 Authoring Systems	96
6.3.2 Generic Training System Model	96
6.3.3 Automatic Knowledge Acquisition	96
6.3.4 Reverse Engineering of Knowledge Bases	96
6.4 Recommendations for Future Work	97
6.4.1 Interactive Simulations	97
6.4.2 Induction Tuning	98
6.4.3 Real-world Testing	98
6.4.4 Complete ITS Development	99
6.5 Overall Conclusions	99

	Page
Appendix A. Definitions	100
A.1 Simulation	100
A.2 Knowledge Base	100
Appendix B. Prototype Algorithms	101
B.1 Scenario Exploration	101
B.2 Induction	103
B.3 Code Availability	103
Bibliography	105
Vita	109

List of Figures

Figure	Page
1. Standard ITS model	2
2. Knowledge communication hierarchy	5
3. Shallow vs. Deep Knowledge	12
4. Operator Decision-Making Schematic	17
5. Component Display Theory	20
6. Knowledge Separation	25
7. Knowledge vs. Training effectiveness	26
8. Knowledge vs. System Cost	27
9. SCIUS Architecture	32
10. An Abstract Simulation	34
11. Hypothetical Simulation State Graph	35
12. An Abstract Knowledge Base	36
13. Rule Induction Flow	50
14. Knowledge Base Flattening	51
15. State vs. Action Based Exploration	58
16. Example Simulation Graph	59
17. Expert Path Deviations	61
18. Consistency Checking	65
19. Simulation Interface	72
20. Simulation Control Interface	75
21. Expert Simulation Exploration	77
22. 1 Generation, 1 Branch Exploration	78
23. Simulation Exploration With (a) 1 Generation, 10 Branch, (b) 1 Generation, 100 Branch	79
24. Simulation Exploration With (a) 10 Generation, 1 Branch, (b) 100 Genera- tion, 1 Branch	80

Figure		Page
25.	2 Generation, 5 Branch Exploration	81
26.	Ideal Timing	82
27.	Induction Accuracy	84

Abstract

Intelligent Tutoring System (ITS) development is a knowledge-intensive task, suffering from the same knowledge acquisition bottleneck that plagues most Artificial Intelligence (AI) systems. This research presents an architecture that requires knowledge only in the form of a shallow knowledge base and a simulation to produce a training system. The knowledge base provides the basic procedural knowledge while the simulation provides context. The remainder of the knowledge required for training is learned through the interaction of these components in a state-space scenario exploration process and inductive machine learning. These knowledge components are used only at the interface level, allowing the internal representation to take any form that meets the interface requirements. A prototype of this architecture is implemented as a proof-of-concept to illustrate the viability of the key knowledge acquisition techniques.

A Generic Intelligent Architecture for Computer-Aided Training of Procedural Knowledge

I. Introduction

If you don't have a gadget called a teaching machine, don't get one. Don't buy one; don't borrow one; don't steal one. If you have such a gadget, get rid of it. Don't give it away, for someone else might use it. This is a most practical rule, based on empirical facts from considerable observation. If you begin with a device of any kind, you will try to develop the teaching program to fit that device. — Gilbert, 1960 (18).

The impact of the computer upon society has been considerable, but the impact upon the educational process has been underwhelming, to say the least. While a print shop owner from the 1800s would be bewildered by the desktop publishing machines of today, a teacher from the 1800s would (mostly) feel right at home in a modern classroom. The main reason teachers have been safe from the computer invasion is that the job of teaching, like many human activities, is much more complex than it seems. The first computer-based teaching systems were little more than electronic workbooks – a pale shadow of the human teacher's capability. Modern research into Artificial Intelligence (AI) holds promise to close this gap somewhat, but at a significant cost. This research is concerned with the task of making intelligent computer-based training systems easier and less costly to build.

1.1 Background

The field of Intelligent Tutoring Systems (ITS) is one of the newer fields of Artificial Intelligence (AI), and grew out of a general dissatisfaction with the more conventional Computer-Aided Instruction (CAI) learning environments. In contrast to a CAI system, an ITS is generative and adaptive, both in content and form to the individual needs of each pupil; this provides a learning experience that is closer to that experienced with a human teacher. The classic ITS model involves four distinct components: an *expert module* to provide domain-specific knowledge, a *tutor module* to provide pedagogical knowledge,

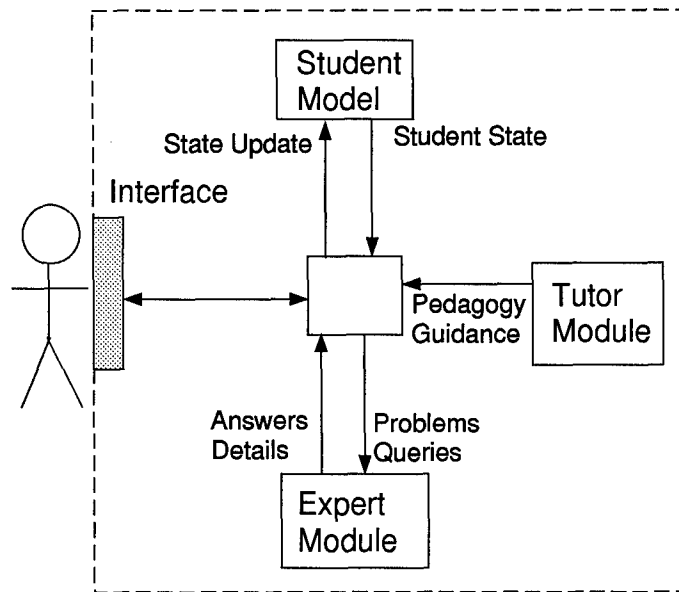


Figure 1. Standard ITS model

a *student model* to provide some estimate of the student's knowledge state, and an appropriate *human-computer interface* to both the domain and instructional components of the system. These components are shown in Figure 1. Researchers have concentrated on various aspects of the basic architecture, typically towards increased complexity – complex student models, innovative pedagogical techniques, large libraries of domain concepts and misconceptions, and adaptive interfaces. While researchers have made some interesting discoveries in understanding students' thought processes and developing innovative instructional techniques, ITSs are still not used in mainstream education.

1.2 Overview

The high-level goal for this research is to make the process of building intelligent training systems less costly. This section describes the problem this research addresses, followed by a motivation for, and description of a unique approach to the problem.

1.2.1 The Problem. Anderson (3) states that traditional CAI development takes 200 hours per hour of instruction, and ITS development could be an order of magnitude greater; Orey (29) describes another estimate of 500 hours development per hour of instruc-

tion. ITS development, like most AI endeavors, is a knowledge-intensive process, requiring a significant amount of engineering to encode knowledge into a form usable by the ITS. This process can be quite complex, expensive and time-consuming; this *knowledge-acquisition bottleneck* is one of the major hurdles for real-world AI systems.

1.2.2 Motivation. The motivation for this research came from a search of the available literature involving Intelligent Tutoring Systems (ITS). Many of the researchers in the field came from primarily educational backgrounds, and as such, were primarily concerned with pedagogical issues. For example, Shute (43) cites the great ITS debates of the 1990s as:

1. How much learner control should be allowed?
2. Should learning take place individually or collaboratively?
3. Is learning situated, unique, or symbolic?
4. Does virtual reality uniquely contribute to learning?

Many of the systems presented in the research are quite complex, involving such concepts as cognitively-based student models, libraries of student misconceptions, natural language interfaces, and generative buggy procedures. Many of the researchers appeared to be more concerned with how much capability can be built into an ITS, instead of how we can make building ITSs feasible in a real-world environment with economic constraints.

Thus, instead of "pushing the frontier" of technology for intelligent tutoring systems, I decided to attack the other end of the spectrum and find out how little could be built into an ITS and still have it behave intelligently. Knowledge is power, but knowledge is also expensive. Feigenbaum's knowledge acquisition bottleneck (15) plagues AI systems and can only be attacked by reducing the knowledge engineering required for a system.

1.2.3 Approach. This research concentrates on training domains, involving the communication of procedural knowledge to operators of complex dynamic systems. The goal of this research is to develop a domain-independent architecture for an intelligent training system that minimizes the amount of domain knowledge engineering required for a complete system, and takes advantage of pre-existing knowledge, if available. The

domain knowledge in the architecture is isolated in two components: a simulation of the domain and a knowledge base that is required to contain only shallow knowledge about the operator's job.

However, these two components alone, even in a generic architecture, are not sufficient for intelligent training. Instead, the remainder of the knowledge required for training is developed automatically through the interaction of these components. More specifically, the scenarios required for student practice are discovered through an automated state-space search process. The knowledge base provides the *what* for a particular scenario, but machine learning induction develops the *why* for classes of scenarios, providing a means for generalized training, and as information for developing an automated baseline curriculum. Additionally, these components are accessed only at the interface level, allowing the internal representation of these components to take any form that meets the interface specification. This allows enormous flexibility in the development of the domain knowledge components, as well as a means for utilizing pre-existing knowledge where available.

In brief, a simulation provides context without knowledge; a knowledge base provides knowledge without context. This research exploits the interaction of these components to provide combined knowledge that is more powerful than the union of these components in isolation. This allows the development of an intelligent training approach that requires less knowledge than other approaches, and represents a significant savings in terms of development and maintenance time and effort.

1.3 Scope

This research is applicable to domains involving training of procedural knowledge to adults, as shown in Figure 2. Because this research exploits certain characteristics of this type of knowledge communication, it may not be particularly applicable to other domains, primarily because the other domains have significantly different learning objectives.

Furthermore, this research concentrates on the *knowledge acquisition* problem of intelligent tutoring systems, specifically acquiring a level of domain knowledge sufficient for training under a training model described in Chapter 2. Knowledge acquisition is a

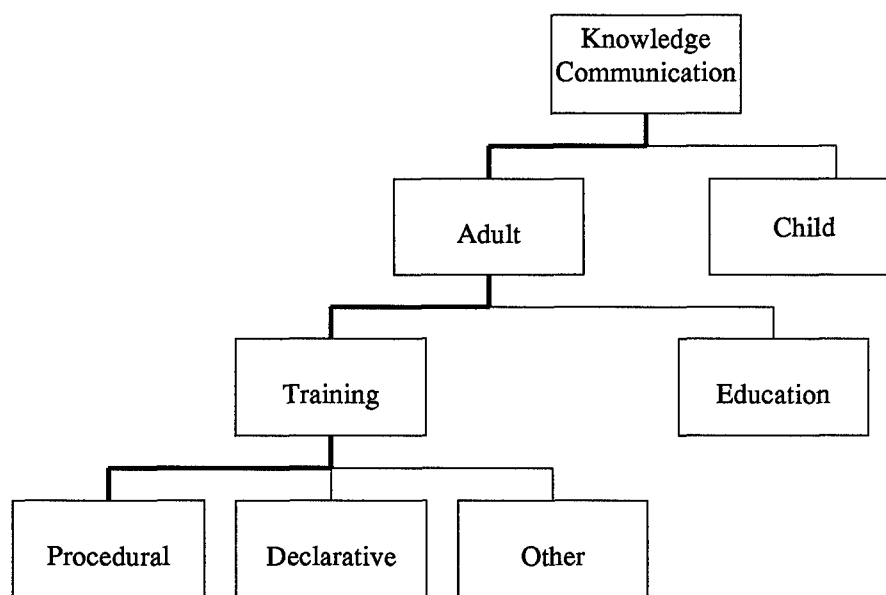


Figure 2. Knowledge communication hierarchy

key factor in the efficiency of ITS development (26). In the development of ITSs, the next step after acquiring knowledge would be to implement the presentation/instructional interaction component of the system. This task can be challenging from a pedagogical perspective, but is mainly labor-intensive from an implementation perspective. For the same reasons, the emphasis in the ITS field has been away from full system development, and has instead concentrated on the key pieces of system development (43). This effort is intended to be a foundation research effort, exploring the feasibility of a unique approach to acquiring knowledge for intelligent training systems.

1.4 Objectives

- Develop an architecture for a training system that uses only a simulation and knowledge base for domain knowledge.
- Develop an approach to automatically generate scenarios for training.
- Develop a technique to isolate the key feature knowledge from a scenario.
- Determine the knowledge and interface requirements for a simulation and knowledge base in a generic architecture.

These objectives are parts of an overall vision, but they are largely independent in potential applicability. For example, the automatic generation of scenarios may be applicable to other ITS research efforts, whether or not they use my overall approach.

1.5 Research Structure

This study first develops a methodology and architecture for a generic training system, followed by the development of algorithms and techniques for automatic knowledge acquisition of additional domain-dependent training knowledge. The knowledge-acquisition components of the architecture are then implemented as a prototype on a PC-compatible workstation under Common Lisp. Finally, the architecture is demonstrated in a specific domain as a proof of concept.

II. Background

This chapter discusses the relevant concepts and issues in the field of Intelligent Tutoring Systems (ITS), as well as some related fields that apply to my research, followed by a discussion of related research efforts in the field.

2.1 History

The field of computer-aided instruction (CAI) began in the early 1950s as a realization of the "programmed learning" movement of the period. The philosophy of instruction was based upon Skinner's (47, 46, 20) behaviorist theories, and the general concept was to reduce a domain to primitive chunks which could be taught to a pupil individually. Each chunk was a prerequisite for other chunks, and had prerequisites itself, forming a directed graph of subject material. This strict linear progression through chunks forced the student to follow the structure imposed by the author of the tutorial (56, 13).

In the 1960s, it was realized that student feedback could provide some additional control to a specific student's progression through the material. At first, this was simply implemented as a set of test questions at the end of each subject chunk, controlling advancement to the next chunk. If the test questions were not passed, then the student could be sent back to review the material again, or the student could branch to some form of corrective feedback. At this time, the branching was explicitly controlled by the program's author, and student responses outside the author's pre-programmed branching scheme were not possible. Although these systems were more adaptable than early systems, they still forced the student to follow a strict, planned learning progression.

In the 1970s, "authoring languages" were developed to allow non-programmers to develop CAI systems. An authoring language is essentially a domain-independent template that controls the structure of the CAI system; the author only has to provide the domain subject material. While authoring systems make the development of CAI systems easier, the trade-off is that they significantly limit the flexibility of the instruction. The tutorial author is forced to follow the authoring system's template of how a generic lesson should be structured. Also in the 1970s, the first generative tutorial systems were devel-

oped. In some simple domains such as math, the system could generate its own problems (addition, subtraction, etc.), solve them, and adjust the difficulty to match the student's ability (mostly by creating bigger or smaller arithmetic problems). Typing tutors were also developed that could adapt the typing drills to concentrate on letters that were causing the student trouble.

Basic CAI systems have obtained significant gains over the programmed learning machines of the 1950s, but they are certainly no match for a human teacher. In CAI systems the computer is acting as a presentation device, even though sometimes the presentation can be made somewhat complex by means of the pre-defined branching strategies already discussed (13). The essential problem is that CAI systems operate under severely impoverished knowledge; "none of them have any human-like knowledge of the domains which they are teaching; nor can they answer serious questions of the students as to 'why' and 'how' the task is performed" (56).

The field of Intelligent Tutoring Systems (ITS) began as an attempt to deal with the shortcomings of basic CAI systems. It was felt that the growing field of AI could provide the CAI systems with some human-like reasoning abilities, thus greatly increasing the effectiveness of the instruction. In contrast to CAI, an ITS is generative and adaptive, both in content and form to the individual needs of each pupil. Also, the range of adaptations is not explicitly planned in advance, as in a simple branching CAI system; the adaptation progresses as the interaction with the student progresses (13).

Computers have long held the promise of revolutionizing education. The idea of an infinitely patient, highly knowledgeable, one-on-one private computer tutor has often been touted as the future of education. However, after forty years of research involving computers and education and development of over 10,000 pieces of educational software, traditional methods of teaching are still dominant. Because teaching is such a dynamic and knowledge-intensive process, AI may hold the only potential for realizing any of the long-term goals of computer-based instruction (3).

2.2 Intelligent Tutoring Systems

There are different viewpoints on the precise components of an ITS, but one widely accepted definition is by Wenger. Wenger's three main components of an ITS are: an expert module that represents domain expertise, a student model that represents the student's knowledge state and expertise, and a tutor, or pedagogical module that structures the interaction between the tutor and the student (54).

2.2.1 The Expert Module. The expert module serves two primary functions. First, it acts as a source for knowledge, which includes generating intelligent responses to student queries as well as intelligent tasks and questions for the student. Second, it must act as a standard for evaluating the student's knowledge and performance; it must be able to solve problems in the same context as the student, so their respective answers can be compared (54).

Anderson defines three basic types of expert modules (2). The first is the *black box* model, that represents domain knowledge in a manner that is completely different from human reasoning. For example, SOPHIE reasons about electronic circuits using the SPICE simulator and mathematical relaxation (6). Essentially, it uses a numerical process to achieve results similar to what humans achieve with a symbolic process (2). The second type of expert module is the classic expert system, which may or may not closely represent human reasoning processes, (Anderson calls these *glass box* systems). GUIDON was a system designed by Clancey (9) that made extensive use of the classic expert system MYCIN (41). The third type of expert module is the higher level *cognitive model* type. This type of module is essentially a simulation of some form of human reasoning. The knowledge in these systems can take many forms such as qualitative simulation (21), semantic nets (7), and Socratic dialogue (49).

2.2.2 The Student Model. Intelligent communication requires some understanding of the recipient (54). Therefore, an ITS must have some understanding of the student's current state of knowledge. The process of inferring a student model is often referred to as *diagnosis* because the process is similar to medical diagnosis; an ITS uncovers a hidden

cognitive (as opposed to physiological) state from observable behavior (52). Basically, a student model is required by the ITS to organize and guide the student's learning process (54).

VanLehn describes a three-dimensional space of student models based upon bandwidth, target knowledge type and the differences between the student and expert (52). The first dimension, bandwidth, is concerned with the amount of the student's activity that is available to the model. Anderson's LISP tutor (35) was an example of a high bandwidth model that contained a detailed cognitive model, representing a sequence of student mental states (in Anderson's view). In Anderson's tutor, the student was required to pick menu selections to represent goals, strategies and code fragments to use (52). On the other hand, PROUST was a low bandwidth system designed to teach Pascal programming. PROUST only used the final program submitted to the compiler as a source of knowledge for inferring a student model (52, 54). VanLehn's second dimension, knowledge type, deals with whether the knowledge to be taught is procedural or declarative (procedural and declarative knowledge are discussed in Section 2.4.1). If the knowledge is procedural, it may be flat or hierarchical. The type of knowledge affects the modeling process; most systems use a mixture of procedural and declarative, even when the domain is primarily of one type. The third dimension involves how the difference between the student module and the expert module is represented. Some systems simply represent the student as a subset of the expert; this type of model is referred to as an overlay model. These systems find *missing conceptions*, or items of knowledge the expert has that the student does not. Other types of systems find *misconceptions*, or items of knowledge the student has that the expert does not. In these systems, the misconceptions are often represented as malformed expert rules (malrules) or "buggy" rules. Ideally, such a system will have a large number of bug rules to cover most of the potential student misconceptions. These buggy rules can be obtained from a pre-compiled bug library or generated through some type of learning theory (52, 5). Systems which use this model must be able to match the student's actions to a buggy procedure in the bug library, or be able to generate a buggy procedure that gives the same incorrect result as the student.

2.2.3 The Tutor Module. The purpose of the tutor module is to define the pedagogical strategy, or the organization, sequencing and form of the tutor-student interaction. Essentially, the tutor module embodies "knowledge about communicating knowledge" (54).

The tutor module takes a wide variety of forms in different ITS systems. In some systems, the pedagogical knowledge is deeply embedded in the code controlling the interaction with the student (54). In other systems, the pedagogical knowledge is represented explicitly and separated from the rest of the tutor (9, 8). Tutor modules have to make decisions at both the global level, controlling the sequence of instructional units, and the local level, involving guidance, explanation and remediation with the student during a single tutoring session. ITS systems also vary in the degree of control they exert on the student-tutor interaction. Systems range from the highly structured traditional CAI systems, to guided-discovery systems (42), to pure discovery systems such as LOGO (30).

2.3 Related Concepts

This section provides a background to concepts that are not directly related to the ITS field, but are important to this research effort.

2.3.1 Shallow vs. Deep Knowledge. In terms of knowledge-based systems, the "depth" of knowledge refers to the degree of sophistication in the relationship between cause and effect.

An example of shallow and deep knowledge is shown in Figure 3. Both representations encompass the relationship between the accelerator and the speed of the car, but the deep representation contains much more causal information. The main trade-off between shallow and deep knowledge is explanation capability versus efficiency. Thus, shallow knowledge is more efficient and easier to engineer, but it is difficult (or impossible) to justify the decisions made by a shallow knowledge system.

2.3.2 Simulations. Shannon describes a simulation as:

the process of designing a model of a real system and conducting experiments with this model for the purpose either of understanding the behavior

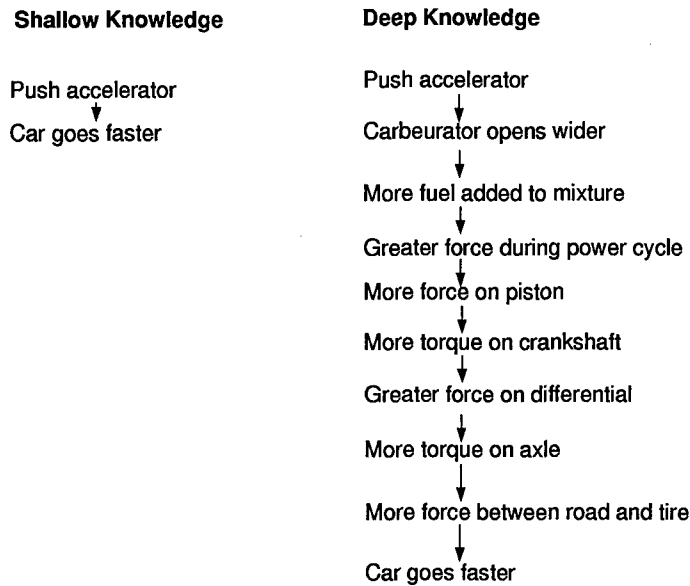


Figure 3. Shallow vs. Deep Knowledge

of the system or of evaluating various strategies for the operation of the system (39).

A number of widely variant models can be classified as simulations, but one distinction among simulations is the division between *interactive* and *non-interactive* simulations. Non-interactive simulations take some initial set of parameters and run deterministically to some finish state with no further input. Interactive simulations, on the other hand, also have an initial set of parameters, but can take further input during the execution of the simulation. Simulations can also be *time-based*, or *event-based*. Time-based simulations have some internal periodic clock, and the simulation objects change state based upon the time interval used for the clock. Event-based simulations have a variable clock which is controlled by the interactions of objects in the simulation; if there are no object interactions within the simulation during a period, then the simulation clock "jumps" ahead, and conversely the simulation clock increases in granularity if there are many closely-spaced object interactions. A time-based simulation will always take the same amount of wall-clock¹ time to simulate a given amount of simulated time. The wall-clock time needed for

¹Wall-clock time is a term used to distinguish the time required to run a simulation from the time being simulated. For example, it may take several days of wall-clock time to simulate a microsecond in a computer, or it may only take an hour of wall-clock time to simulate an epoch in a geological simulation

an event-based simulation will vary based upon the number and interactions of the objects within the simulation.

Event-based simulations are typically used for high-performance applications because they can often run significantly faster than an equivalent time-based simulation. However, for interactive simulations, the time-based strategy is chosen. This is because the internal clock in an interactive simulation must be “synched” to the wall-clock. For example, a user running a driving simulation will expect that a second of wall-clock time roughly correspond to a second of simulation time.

For the purposes of this research, a simulation will be considered to be a model of some real-world system that an operator might control. The simulation will be composed of a number of objects that correspond to the objects of which an operator would be aware. Ideally, the interface to the simulation is as close as possible in appearance as the interface to the real-world system. The objects have various state variables, and the combination of all the state variables in the simulation comprise a simulation *state vector*. The simulation will be discussed in more detail in Chapter 3.

Using this restricted definition of a simulation, the simulation can be modeled as a particular type of finite-state automaton called a *transition graph*.

2.3.2.1 Transition Graphs. A transition graph is a mathematical model that is composed of three things (10):

1. A finite set of states, at least one of which is designated as the initial state, called the start state, and some (maybe none) of which are designated as final states.
2. An alphabet Σ of possible input letters, from which are formed strings, that are to be read one letter at a time.
3. A finite set of transitions that show how to go from one state to another based upon reading specified substrings of input letters (possibly even the null string Λ).

Applying this model to a simulation, the first component is the set of all possible state vectors (the state space) in the simulation. The second component is the power set² of all possible control inputs (the control space) to the simulation, plus the time input. The

²The power set of a given set is all possible combinations of the elements of the set taken one at a time, two at a time, ... up to n at a time.

third component is the simulation itself, which controls transitions between states based upon control inputs and time.

It is worth mentioning that although a transition graph and a simulation are both finite, the real world is infinite. Any computer model of the real world will be ultimately finite, and an approximation of the real world.

2.3.3 Machine Learning. Machine learning is a sub-field of Artificial Intelligence that is concerned with giving a system the capability to improve its performance over time, typically through some automatic creation or acquisition of knowledge. Machine learning covers the spectrum from completely constrained, guided (rote) learning to completely unguided discovery-based learning approaches. This research uses *inductive learning*, which is primarily concerned with the automatic generation of a hypothesis that distinguishes among positive and negative examples of a target concept. Typically, an inductive system is “trained” on a set of examples, a hypothesis is generated, and then the hypothesis is tested on a set of unseen examples to test its generality. One of the most well-known and extensively studied examples of an inductive learning system is Quinlan’s ID3 (32), which used a decision-tree approach to classify positive and negative examples. A version of ID3 is used for this research.

2.4 Educational Theory

Standard, widely accepted theories of learning and training were used in the design of the system. The following sections describe the relevant theories used in the areas of training, learning, decision-making and practice.

2.4.1 Learning Theory. To be able to train an operator, some understanding of the human learning mechanism is required. For this research, a well accepted theory of learning was selected based upon its applicability to the general task of training; this theory was then used in the design of the overall system architecture. This section describes the theory chosen. Later sections describe how it is used in this particular research.

The main learning theory used is Anderson's ACT* theory (1). One of the central elements of Anderson's theory is the distinction between *procedural* and *declarative* knowledge. Declarative knowledge is knowledge that is factual in nature, and can be made explicit, whereas procedural knowledge is knowledge about how to accomplish some task such as driving a car or troubleshooting a circuit (31). Basically, Anderson's theory states that skill acquisition occurs when declarative knowledge is converted, or compiled to procedural knowledge (productions) through practice. Anderson justifies the requirement for this compilation process in terms of the adaptability of the human cognitive system. Procedural knowledge controls behavior, thus it must be tested out and proven before it is internalized.

The first step in Anderson's skill acquisition theory is the *declarative* stage. During this stage, the trainee has only declarative knowledge about the task he is learning. This knowledge may take the form of basic facts about the domain, general-purpose procedures, or specific procedures. However, even if the trainee knows specific procedures about the task, this is not considered procedural knowledge because the trainee is using this knowledge interpretively, until it is used and compiled to procedural knowledge through practice (1). Anderson clearly makes this distinction:

The acquisition of productions is unlike the acquisition of facts or cognitive units in the declarative component. It is not possible to simply add a production in the way it is possible to simply encode a cognitive unit. Rather, procedural learning occurs only in executing a skill; one learns by doing (1).

The advantage of using declarative knowledge interpretatively in the first stage of skill acquisition is flexibility. However, interpretation is costly in terms of human memory and speed, so the knowledge must eventually be compiled into a procedural form.

The second step of the skill acquisition process, *knowledge compilation*, encompasses two processes: compilation and proceduralization. Compilation is the collapsing of a sequence of productions into a single production that has the same effect as the sequence (chunking). Proceduralization is the instantiation of variables in a production, to essentially create a more specialized production, thereby eliminating some of the demand on long-term memory retrieval (1). The proceduralization process is analogous to Shiffrin's

automatization process (34, 40), whereby the performance of skills is learned in an automatic fashion, eliminating most of the cognitive load required for performing the skill.

Once the declarative knowledge has been converted to procedural knowledge, it is further refined in the third stage of skill acquisition: *tuning*. The tuning stage of skill acquisition consists of three phases: generalization, discrimination, and strengthening. Generalization is essentially the process of replacing bound facts in a production with variables to broaden the production's scope of applicability. For example, this can have the effect of eliminating productions when two or more productions have identical consequents, and the generalization process results in the productions having matching antecedents. Discrimination is the addition of antecedents to a production, which has the effect of narrowing the scope of the production. Finally, strengthening is the process whereby competing productions are weighted based upon feedback as to their applicability (reliability). Anderson theorizes that positive feedback is a more gradual process than negative feedback; in other words, a production gets slowly promoted over time as it proves correct, whereas a production will be quickly demoted if it proves incorrect.

Anderson's theory is recognized both in the cognitive science field (31, 28, 24) and the ITS field (43, 54, 52, 51, 16, 55). Shute's criticism of Anderson's theory is that it is more applicable to procedural type domains, and less applicable to ill-structured domains such as creative writing or economics (43). However, Shute's criticism clearly supports the use of Anderson's theory in a architecture designed for training.

2.4.2 Operator Decision-Making Theory. The main job of the operator of a Complex Dynamic System (CDS) is to make decisions and perform actions based upon the state of the system he controls. I felt that it was important to use a sound theoretical framework of the operator decision making process to facilitate the appropriate design of the research architecture. This section describes the decision-making theory selected.

Rasmussen (33) has presented an extensive analysis of the information-processing processes involved in CDS operator decision-making. Of particular interest is the *information processing schematic*, reproduced in Figure 4. The map shows a sequence of information processes that an operator might use in a control decision. A theoretical op-

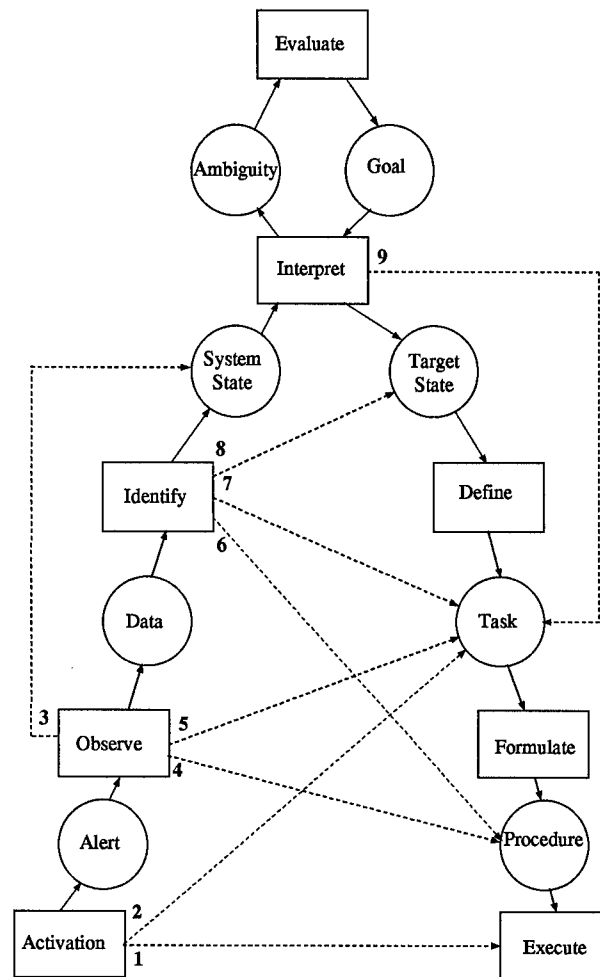


Figure 4. Operator Decision-Making Schematic

erator could pass through all information processing blocks (solid arrow path), but often “short-cuts” are taken, as indicated by the numbered, dotted lines. The information blocks in the figure require some further explanation (adapted from (33)):

- Activation – This is the initial trigger that some action needs to be performed. It results in a binary “alert” information state.
- Observe – This is the process whereby some set of observations are made to gain data about the system.
- Identify – This is the process where the observed data is analyzed to produce an overall system state determination.
- Interpret – This is the step involving high-level thought about the consequences of the current state, safety considerations, efficiency, etc.

- Evaluate – This is another high-level thought step that compares the overall performance criteria against the current considerations to establish an ultimate goal based upon the current decision.
- Interpret II – This process “grounds” the high-level goal back down a specific target state for the system.
- Define task – The process determines which activities are required to reach the target state.
- Formulate procedure – In this step, a plan is generated to accomplish the desired task.
- Execute – The proper action is performed.

Clearly, every operator decision does not involve this entire process, the shortcuts are frequently used for a variety of reasons (adapted from (33)):

1. This is strict stimulus-response behavior, involving a single action.
2. This shortcut is used when time constraints require a fast response; it is similar to stimulus-response behavior, but involves a more complex sequence of actions.
3. This shortcut is taken when a subset of the overall system state is considered as an overall indication of the system state, eliminating the need to evaluate the data in terms of the entire system.
4. This shortcut is used when the student has learned a pre-defined cue for a pre-defined procedure; this is essentially shallow knowledge for a procedure.
5. This shortcut is taken when certain data is a cue for a set of actions that must be formulated into a procedure; this is one step deeper knowledge than the previous shortcut.
6. This shortcut is analogous to number four, and is used when a pre-defined procedure is cued by an overall system state.
7. This shortcut is analogous to number five, and is used when a set of tasks is cued by an overall system state, but the tasks must be converted into a procedure.
8. This shortcut skips the determination of a system state in favor of the determination of the desired target state.
9. This shortcut is used when the considerations of the current state directly imply a set of tasks, without the need for deeper evaluation against some overall system criteria.

It is worth noting that only one of the shortcuts leads to a “non-action” state, the other eight all lead to task, procedure, or execution states, which are all aspects of the same concept of “operator action.” This aspect of the decision-making schematic is exploited in the design of this research, and will be explained in a later section.

Rasmussen (33) divides his map into three main types of human operator behavior: skill-based, rule-based, and knowledge based. Skill-based behavior involves only the activation-execution shortcut. Knowledge-based behavior involves the interpret-evaluate cycle, while rule-based behavior involves everything in-between these two regions. My research is almost exclusively concerned with rule-based behavior, with some overlap into skill-based behavior. Knowledge-based behavior is a completely different problem, and is not addressed in my research.

2.4.3 Training Theory. As previously described, the goals and processes involved with training vs. education are significantly different, with education admittedly more complex than training. Many educational theories that may be considered too simplistic to apply to the broad scope of "education" can be adapted to the more limited scope of "training" quite well. This section describes some of the theories used and adapted for this research.

2.4.3.1 Gagne's Instructional Theory. One of Gagne's most important contributions to the field of educational theory was the fundamental idea that all learning is not the same, and that varieties of learning differ based upon the *conditions* that are required to promote a particular type of learning (31). Gagne proposes a taxonomy of learning activities that range from *stimulus discrimination* at the low end (cognitively) to *learning schemata* at the high end. My research is primarily concerned with the range in Gagne's hierarchy between "response learning" and "response integration." According to Gagne, the best method for facilitating learning in this range is practice with feedback, and suggestive information on how to make a proper response (31). A second aspect of Gagne's theory is his nine instructional events (17), which can serve as a general-purpose pedagogical structure for presenting information to the student:

1. Gaining attention
2. Informing learners of the objective
3. Stimulating the recall of prior learning
4. Presenting the stimulus
5. Providing "learning guidance"

6. Eliciting performance
7. Providing feedback
8. Assessing performance
9. Enhancing retention and transfer

Level of Performance				
FIND				
USE		✓	✓	
REMEMBER	✓	✓	✓	
	FACT	CONCEPT	PROCEDURE	PRINCIPLE

Figure 5. Component Display Theory

2.4.3.2 Merrill's Component Display Theory (CDT). Merrill's Component Display Theory (CDT) (27) provides a framework for content and goals of knowledge used in training. Figure 5 shows Merrill's performance-content matrix. *Remembering* involves the simple retrieval of stored information. *Using* involves the application of concepts to new situations. *Finding* involves the discovery of new concepts. The two shaded areas in Figure 5 represent the idea that facts are too basic to be "used" or "found." This may seem unusual, since one can conceive of situations where facts can indeed be used and found. However, this is a terminology confusion; under Merrill's usage "fact" equates with what many consider a "concept." The checked blocks in the figure represent areas that are pertinent to procedural knowledge training. As an example, the various types of knowledge used in the domain of automobile driving might be:

- Remember-fact: What is the maximum safe speed on an icy road?
- Remember-concept: What are the characteristics of a skid?
- Use-concept: If your car is traveling sideways, are you in a skid?
- Remember-procedure: What is the best way to recover from a skid?

- Use procedure: Demonstrate a skid recovery

Using this framework of knowledge, Merrill postulates four different types of training material presentation strategies (31):

- Expository general (telling a rule)
- Expository instance (telling an example)
- Inquisitory general (asking about a rule)
- Inquisitory instance (asking about an example)

Both the knowledge framework and presentation strategies are used in the instructional design strategies for my research. The specific details of how these theories are applied are discussed in a later section.

2.4.4 Practice Theory. Because this research is primarily concerned with training, and because practice is a primary factor in the training experience, a theoretical justification for the level of practice required for a task was desired. Unfortunately, practice theory is not as well developed as education and training theories. However, some use can be made of the theories that have been developed.

DeJong (36) postulated that the time to perform a repetitive task reduces exponentially until it reaches some limit beyond which no improvement is possible. This relationship has been verified empirically (11). The important aspect of this theory is that improvement can be expected to be rapid initially, and then level out to a point where further practice is not beneficial because of diminishing returns. Shute et. al. (44) verified this result using a more general performance measure than cycle time. Although they did not develop a formal equation-based measure of practice requirements, they did find that there was an optimal level of practice, and that practice above that level had little or no effect on performance. Their findings suggest that even a minimal amount of practice can be effective, if the performance of the student is closely monitored (as it is in a ITS system), allowing for "just enough" practice to be used.

Another interesting finding with respect to the theory of practice was found by Schmidt and Bjork (38). They found that structured practice, which was optimal for

in-training performance, did not work well with respect to long-term retention and performance. In a broad set of experiments covering a variety of domains, they found that random ordering the structure of practice and feedback produced better long-term performance, but poorer short-term performance. They hypothesize that varying the structure of the practice forces the student to work harder, and isolate the training material from the feedback process, which resulted in better long-term performance of the practiced task. Their results suggest that there is no optimal practice ordering, and in fact a traditionally optimal ordering may be sub-optimal with respect to long-term performance. This also suggests that in-training feedback as to the effectiveness of practice ordering does not predict future long-term performance.

2.5 Related Research

This section discusses some current research efforts that are similar in scope and intent to my research. Many of these efforts are long-term projects (five to ten years). The research projects are described and contrasted with my research approach.

2.5.1 ACQUIRE-ITS. ACQUIRE-ITS (37) is essentially a knowledge-base building tool combined with an ITS authoring tool. After the knowledge engineer builds a knowledge base using the ACQUIRE tool, he can then use the completed knowledge base along with the ACQUIRE-ITS tool to create an ITS based upon that knowledge. The ITS uses a case-based presentation methodology, using the structure of the rules to create cases. Basically, the student is queried about the consequent of various rules based upon the preconditions of the rules. The interaction with the student is text-based, although supporting video and audio can be manually added as part of the ITS development process.

This research is loosely related to mine, in that it presents a "semi-automatic" use of a knowledge base as a domain expert for an ITS. However, in contrast to my research, ACQUIRE-ITS depends upon the particular structure and implementation of the knowledge base, where I only specify an external interface. Furthermore, ACQUIRE-ITS attempts to teach procedural knowledge in a declarative fashion. Because it lacks a true practice component, ACQUIRE-ITS may not effectively teach procedural knowledge; pro-

cedural skills can only be learned by doing, they cannot simply be encoded as a new cognitive unit (1).

2.5.2 ITSIE – Intelligent Training Systems in Industrial Environments. Primarily, ITSIE is an effort to investigate intelligent training systems for industrial domains, particularly the use of qualitative modeling techniques and multiple models of instruction. In support of this, the researchers are also developing a specification methodology, a generic architecture, and a set of construction tools for intelligent training systems (45). The development differs from a classical authoring system in that it supports multiple methods of knowledge representation and instructional techniques. Sime (45) has developed an extensive specification methodology, and some particularly interesting models of operator behavior and knowledge based upon Rasmussen's (33) theory. This has been adapted for use in my system, and is described in a previous section.

The ITSIE effort differs from my research in that it is primarily an authoring system, albeit a complex and comprehensive system. It is an improvement over conventional authoring systems in that it does not restrict the course author to one particular knowledge representation or instructional strategy, but it does restrict the course author to one of the implemented strategies. This may or may not be a problem, depending upon whether one of the implemented strategies matches the needs of potential user. Use of the system requires the author to encode his knowledge into the ITSIE format, including the simulation, domain knowledge and pedagogical knowledge. My approach, in contrast, simply imposes an external interface specification, without any regard for the internal format of the knowledge.

2.5.3 NASA's Intelligent Computer-Aided Training Authoring Environment.

The purpose of this effort is to develop a tool set (authoring environment) that implements the standard ITS model, as well as a means to transition NASA's Intelligent Computer-Aided Training (ICAT) technology to the private sector (53). The architecture is defined in a modular fashion, specifying which modules are required for an ICAT without specifying how those modules are implemented (53). The system is designed to allow instructional designers with limited programming experience to easily encode knowledge about mis-

conceptions, course design, exercises, and interfaces into each component of the ICAT architecture. Additionally, the architecture is designed to take advantage of simulated work environments (SWE) built for other NASA applications.

This research has some commonalities with my research goals, namely the specification of modules at only the interface level, and the desire to use pre-existing simulations of the domain environment. However, it differs from my research in that it is primarily an authoring environment. The instructional designer must completely specify all exercises that are presented to the student, as well as define a complete misconception library that is used for diagnosing the cause of student errors.

2.5.4 Intelligent Simulation Training System (ISTS). The ISTS effort is a joint project between the University of Central Florida, Embry-Riddle University, and General Electric to build a generic training system that can train without the continuous involvement of a human instructor. The researchers have attempted to separate the domain-specific information from a training system, giving a "generic" training system that can input domain-specific knowledge during initialization (4). The domain knowledge present in the system is contained in the Domain Expert (DE), the Domain Expert Instructor (DEI), and the simulation (22). A human instructor must encode knowledge into the system about how to teach in the domain, lesson information, and curriculum information. Additionally, dynamic scenario generation is accomplished by having the human domain instructor define how to change the difficulty of scenarios.

This research is closest to mine in intent, although there are still some significant differences. Similar to my research, the goal of ISTS is to build a truly generic system that requires only a limited amount of domain-specific knowledge upon initialization. However, they require more knowledge than my system in the form of the DEI, which requires significant knowledge engineering upon the part of the human instructor. Furthermore, ISTS's pre-defined approach to dynamic scenario generation also requires more knowledge and limits the flexibility of the scenario generation as compared to my strategy. Finally, unlike my research, ISTS does not seem to implement any form of automatic knowledge acquisition for the domain-specific knowledge, which is a further departure from my approach.

III. Architecture

There are two main tasks required for building an ITS – the construction of the ITS framework, and the engineering of the domain-specific knowledge, both of which are quite expensive in terms of time and resources. This chapter presents a high-level description of a new architecture for a system that addresses both of these problems in training domains. The details of how this architecture operates will be covered in Chapter 4 and 5. The strategy used to attack the first task was to separate the domain-specific components from the domain-independent components in a training ITS, allowing the construction of a “generic” framework into which a course designer could input domain-specific knowledge in a modular form to complete the system. However, instead of requiring the domain-specific knowledge to take some particular form, as in an authoring system, knowledge is only specified at the *interface* level. This presents several challenges, but also has some significant advantages because it allows domain-specific knowledge to be implemented with virtually any tool, in virtually any form. This unique configuration is shown conceptually in Figure 6.

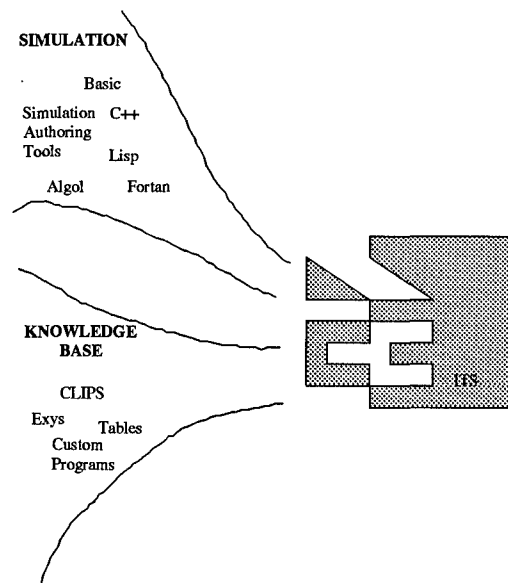


Figure 6. Knowledge Separation

The simulation and domain knowledge represent the domain-specific components of the architecture, the rest of the architecture is generic across virtually any training domain.

These components can be developed using the plethora of tools that are available in these areas, eliminating the need for a specialized authoring system representation and authoring tools. Alternatively, the architecture can take advantage of pre-existing knowledge in either of these components, significantly reducing the knowledge engineering requirements for the system.

The second aspect of ITS building is knowledge engineering. This problem is attacked using two techniques: minimizing the knowledge requirements, and utilizing machine learning for the creation and extraction of new knowledge. These techniques are especially effective in the generic architecture, although they could be used in any architecture.

The first part of this chapter discusses the necessity of the two key domain-specific components of the architecture. Next, an architecture for utilizing these components in a generic fashion is described. Chapter 4 discusses the use of machine learning and scenario exploration used in the architecture.

3.1 Knowledge Requirements

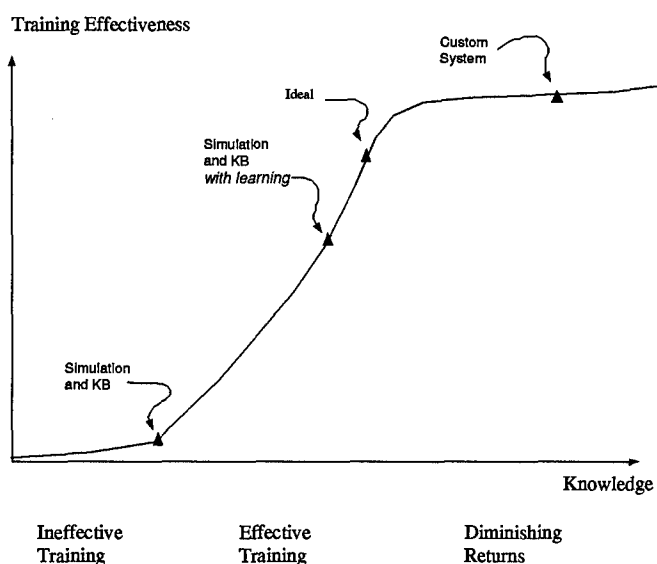


Figure 7. Knowledge vs. Training effectiveness

Figure 7 shows a conceptual graph relationship of the amount of knowledge in an ITS against its training effectiveness. The bottom threshold shows the minimal amount of

knowledge required for effective training. Virtually *any* sort of knowledge communication can be considered at least rudimentary training, but without a minimal level of knowledge, the training is going to be so ineffective or difficult that it is essentially useless. As knowledge is added to this "base level" knowledge, significant gains are made in effectiveness compared with the amount of knowledge added. However, at some point, the idea of "diminishing returns" takes over and additional knowledge has little added effectiveness. The ideal location is at the knee of the curve, where the maximum amount of benefit for the added knowledge has been obtained. Machine learning can act as a "knowledge multiplier," essentially pushing the simulation/KB point farther up the curve of increasing effectiveness.

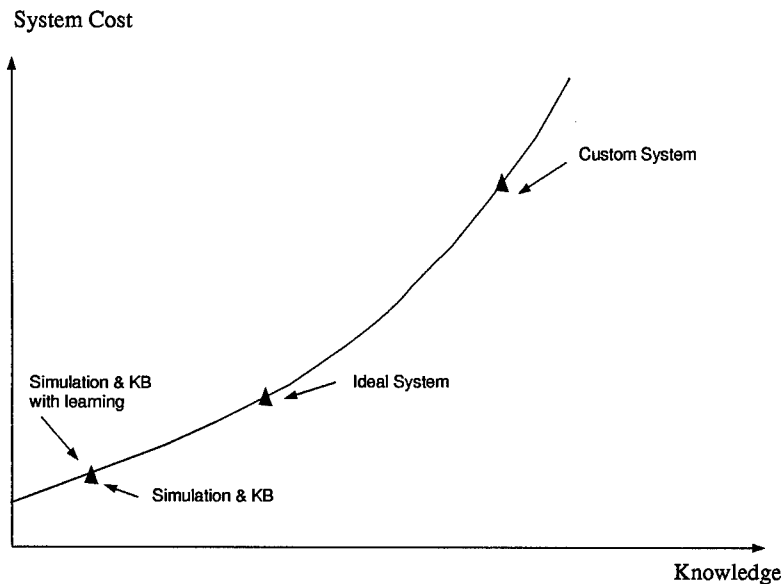


Figure 8. Knowledge vs. System Cost

The ability to automatically expand knowledge through machine learning is particularly attractive from an economic point of view, as shown in Figure 8, where the conceptual relationship between knowledge engineering and system cost is portrayed. The point of this graph is to show that as the amount of knowledge engineered into a system increases, the life-cycle cost of the system increases at a greater rate. This is because the potential conflicts between knowledge and cost/complexity of engineering and retrieving knowledge multiply as the amount of knowledge in the system increases. The interesting aspect of

this relationship is that the addition of machine learning to a simulation and knowledge base increase the effectiveness of the training, but with no corresponding increase in the cost of the system.

There are two aspects to the use of a simulation/KB combination as the fundamental knowledge input for a training system. Two possibilities exist: a simulation and/or knowledge base already exists for a domain, and can be used for a training system, or one/both of these components do not already exist. In the first case, an enormous gain can be realized because a training system can be built with almost no knowledge engineering by using the pre-existing components. However, even in the worst realization of the second case, where both components have to be built, there is still a significant savings when compared to the typical development of a training system.

The remainder of this section deals with the justification for why a simulation and knowledge base represent an amount of knowledge that is at (or below) the minimum level for effective training. In other words, anything *less* than a simulation and knowledge base cannot be considered feasible to use for training.

3.1.1 Training Simulation. There are three primary areas of justification for why a simulation is a necessary part of any intelligent training system – pedagogical, pragmatic, and historical. From a pedagogical perspective, it is clear that ultimately an operator must learn to control some real-world complex system. It is difficult to envision a training program where an operator learns how to control a system exclusively from a book or classroom, then is expected to perform adequately when placed alone in front of the system. Procedural knowledge can *only* be gained by doing (1); this implies that an operator can only truly learn how to control a system by actively controlling that system (practice).

Since operators must learn procedural knowledge by doing, then it may seem that they could just practice on a real-world system. However, real-world systems do not make ideal candidates for training. Frequently, the system that is being controlled is highly expensive or dangerous; operator mistakes can have a devastating effect. Also, most real-world systems have no means of setting up scenarios for the operator to practice, such as

emergency situations. Without that practice, the operator is required to rely on whatever declarative knowledge he may have, which will not be automatized, and may not be reliable. Pragmatically, simulations provide a reasonable compromise for a practice platform. The biggest disadvantage to a simulation is that it is in some sense, an approximation of the real-world system.¹ However, a training simulation may not require an extensive amount of engineering fidelity if it has sufficient psychological fidelity (31). In other words, as long as the simulation behaves externally as the real-world system does, it does not matter how it is implemented "behind the scenes." A training simulation's purpose is to provide a system that mimics the *external* behavior of the real-world system, not to study or model the complexities of the internal physical system that is simulated. Additionally, simulations have several advantages over real-world systems for training. Simulations are typically less costly, less dangerous, can provide accessibility to the full range of possible tasks in a system, and produce less stress on the trainee (31). Finally, simulation "authoring" systems (50) are currently being developed that should greatly simplify the task of building domain simulations for training.

There is ample evidence that simulations are the norm for all intelligent training systems. Historically, there does not appear to be any training system in the literature, either custom made or authoring system, that did not rely on a simulation of the domain for training. An Army study (14) has verified the benefits of simulation-based training.

Simulations appear to be in the mainstream of training system design. Without a practice platform, the trainee is not going to learn how to control the system; he is going to learn declarative knowledge about the system. Pedagogically, pragmatically, and historically, it seems reasonable to assume that a training system without a simulation will not be effective.

3.1.2 Shallow Domain Knowledge. Knowledge is power, so at first it would seem that the more knowledge that can be put into a system, the better. However, like most engineering endeavors, knowledge has a cost – the cost of encoding the knowledge

¹A perfect simulation would be copy of the real-world system. In some situations where the risks are great enough, this type of simulation has been used, such as the command section on the Apollo moon missions.

into the system through the process of knowledge engineering. This cost has become a pervasive problem in the AI field, and was dubbed the "knowledge acquisition bottleneck," by Feigenbaum (15). This bottleneck motivated me to find out how *little* knowledge could be incorporated into a training system and still have it behave intelligently.

To determine the minimal amount of knowledge required for a training system, the first step is to determine the minimal amount of knowledge required to represent the operator's task, and then add the additional knowledge required for a training system. Taken at the most basic level, the operator receives knowledge about the current situation, or state of the system, and outputs a set of control settings (actions) to the system. This implies that the operator's knowledge can be represented, at least conceptually, by a set of "state-action" pairs, where every state of the simulation within his span of expertise² is covered. Thus, any state the system enters (within his span of expertise) has a proper response action.³ The set of state-action pairs represent the *minimal* knowledge required to perform the operator's function, anything less would omit some aspect of the operator's job.

Obviously, most operators do not use a huge set of state-action pairs to accomplish their job, but the state-action pairs are a *model* of the operator's behavior about what to do in any given situation; the two representations are isomorphic. The operator performs as if he had a table of state-action pairs that he was consulting for each decision. Later, I will show that the internal representation is unimportant for the purpose of knowledge representation. Additionally, most operators have some deeper knowledge than that represented by the state-action pairs, but this knowledge is not required to embody the operator's basic function.

Fundamentally, an ideal operator always performs the correct action; his knowledge can be represented as a set of state-action pairs covering his span of expertise in the state space of the simulation. Clearly, anything less than this does not completely represent

²Because the simulation is developed independent of the training system, it may simulate things that are outside of the scope of the expert, and therefore not required for training.

³"Doing nothing" is considered an action.

the operator, thus knowledge isomorphic to the set of state-action pairs is the minimal knowledge required to represent the operator's function.

3.1.3 Summary. A training system without a simulation is training declarative knowledge and not the procedural knowledge necessary for the operator to truly learn the task. A training system without knowledge isomorphic to a state-action pair representation is incomplete, and thus omits knowledge. Without these two components, a training system will be incomplete, inappropriate for training, and likely to be ineffective.

3.2 Architecture

The previous section discussed why a simulation and shallow knowledge are the minimal knowledge requirements for an intelligent training system. The following section will detail my architecture for using these components in a generic training system, called SCIUS⁴ – Self-Creating, Intelligent “Un-authoring” System. “Un-authoring” is a term used to distinguish the SCIUS approach from a conventional authoring system. A conventional authoring system requires a course author to encode the domain specific knowledge into a form compatible with the authoring system. SCIUS, on the other hand, has no such requirement; the domain-dependent knowledge components can be developed without knowledge that they are going to be used in a training system. Therefore, a training system can be built with the SCIUS architecture, essentially *without* a course author.

The key element that allows this architecture to work is the use of machine learning induction (described in detail in Chapter 4). Without induction, a system that used a simulation and knowledge base could only present examples along with their correct actions. It would only be able to explain *what* action was appropriate, but would not be able to explain *why* an action is appropriate. This explanatory information is crucial to effective training because it allows the student to generalize across classes of scenarios, instead of attempting to learn unrelated state-action pairs. Additionally, this information provides a means of structuring the presentation of scenarios to provide a baseline curriculum.

⁴SCIUS is a Latin word meaning “to know.” It is a root of the word “sciolism,” meaning a “superficial display of knowledge.”

The following section describes an overall view of the SCIUS, followed by a detailed description of the components of the architecture.

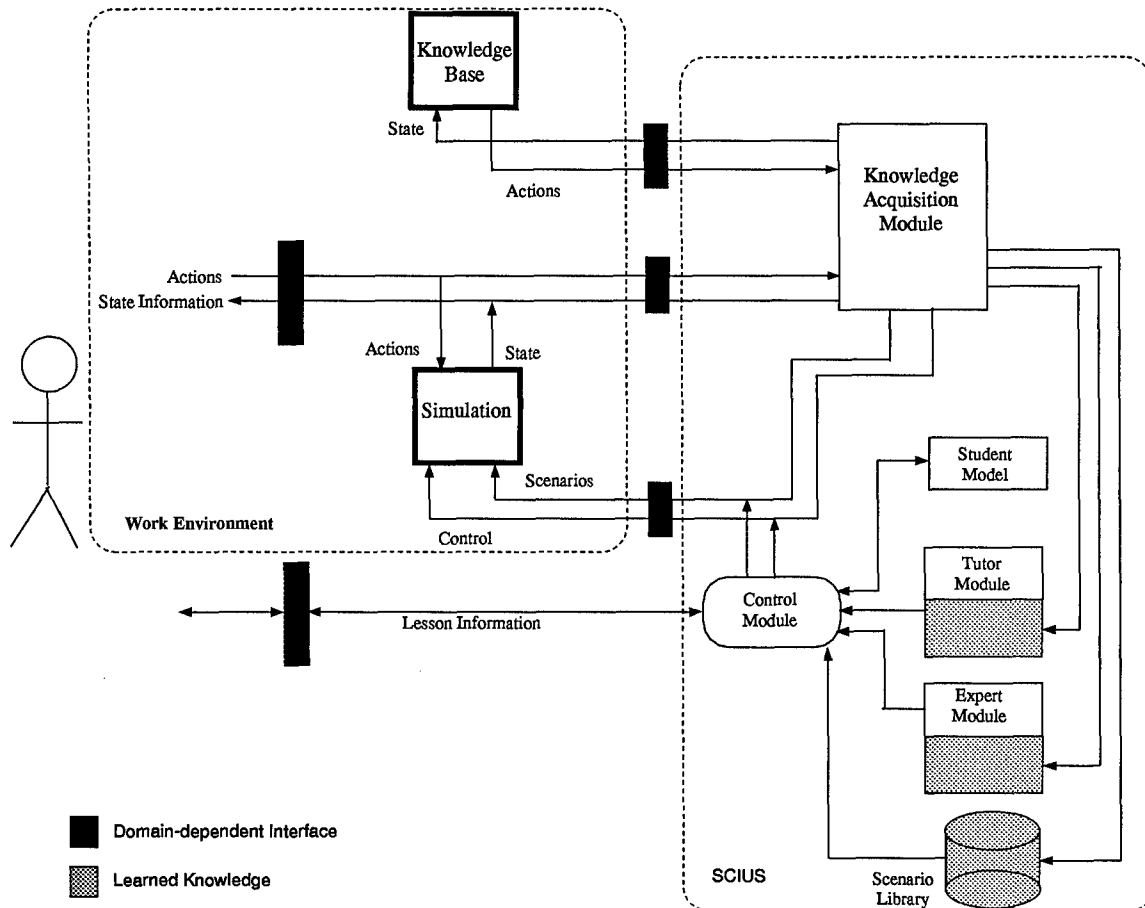


Figure 9. SCIUS Architecture

3.2.1 Overview. Figure 9 shows the overall architecture for SCIUS. This architecture was designed to integrate as seamlessly as possible with the operator's future environment, as reflected in the partition shown in the figure between SCIUS and the work environment. The majority of the training interaction between the student and SCIUS is accomplished through the operator's normal interface with the remainder accomplished through a specialized interface. The black boxes in Figure 9 represent domain-dependent computer interfaces that allow SCIUS to utilize pre-existing knowledge, and human-computer interfaces that allow the student to interact with the system.

The overall operation of the system is fairly straightforward, and is divided into two phases: initial knowledge acquisition and normal training. During initial knowledge acquisition, the knowledge acquisition module (KAM) uses the knowledge base information to explore the simulation space. The knowledge gained during this phase is added to the tutor and expert modules to provide domain-specific knowledge for training. Additionally, a scenario library is built up during this process; these scenarios are filtered, organized, and structured for presentation to the student during training.

During training, the control module uses information from the student model, tutor module, and expert module to control the scenarios demonstrated to the student from the scenario library, as well as control the simulation to allow for scenario presentation. Based upon the student's performance in the scenarios, the control module updates the student model. Additionally, the control module presents additional information to the student with respect to information about scenarios, or other training information through the specialized interface.

3.2.2 The Simulation. The simulation and knowledge base are the two primary domain-specific components of the SCIUS architecture; the remainder of the architecture is generic. Because of this, the simulation and knowledge base are defined in terms of a standard interface; this allows these components to be developed separately from the ITS, or developed without any knowledge that they are going to be used for an ITS.

Fundamentally, a simulation can be represented as a black box that takes input in the form of an action vector, and outputs a new state of the world, as shown in Figure 10. The boxes in Figure 10 represent real-world objects that are represented in the simulation. The gray boxes represent objects that the operator will be aware of, because he directly controls these objects through the action inputs. The operator need not be aware of any of the other objects in the simulation.

Because the action vector can be a null vector, the simulation continually outputs a state vector, one for each time increment of the simulation. Thus, the state of the simulation at any time t is a function of the starting state S_0 , t and the sequence of action vector inputs $\{\vec{A}_1, \vec{A}_2, \vec{A}_3, \dots, \vec{A}_t\}$. In addition to these factors, the state of the

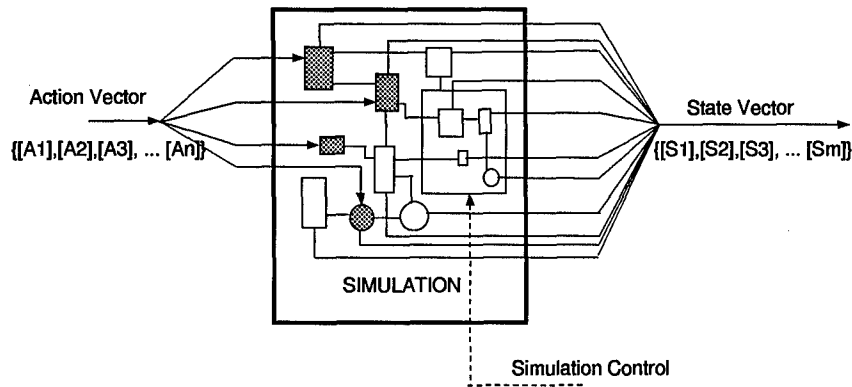


Figure 10. An Abstract Simulation

simulation is also a function of the meta-control vector, which allows the simulation to make transitions that are not part of its normal operation; this will be described in greater detail later. For the purposes of this research the simulation is deterministic, so the state at any time is only dependent upon the factors described above. Non-determinism could potentially be accommodated, but it is an unnecessary complication at this stage in the research.

The simulation can be treated as modeling a transition graph; the nodes in the graph are simulation states, and the transitions between nodes are controlled by the action vector, time, and the control vector. An example of a small simulation state graph is shown in Figure 11. Figure 11 represents part of a graph for a simple driving simulation, with only one state variable, vehicle speed, and two control inputs: brakes {on,off} and accelerator {0% - 100%}. The "120 mph" node represents an illegal state for the simulation, or one that cannot be reached by any other node in the graph. In the case of Figure 11, this might be because 120 mph is above the maximum speed for the vehicle. One unique characteristic of illegal nodes is that the simulation is undefined at these points; transitions from these nodes could go anywhere or nowhere.

In reality, the simulation state graph of any non-trivial simulation would likely be much too large to be represented on paper; if the simulation contained real-valued variables or unbounded integers, the transition graph would be infinite. The number of states in the transition graph is given by $\prod_{i=1}^n s_i$, where s_i is the number of possible values for attribute i , and n is the number of attributes in a state vector. For example, the prototype

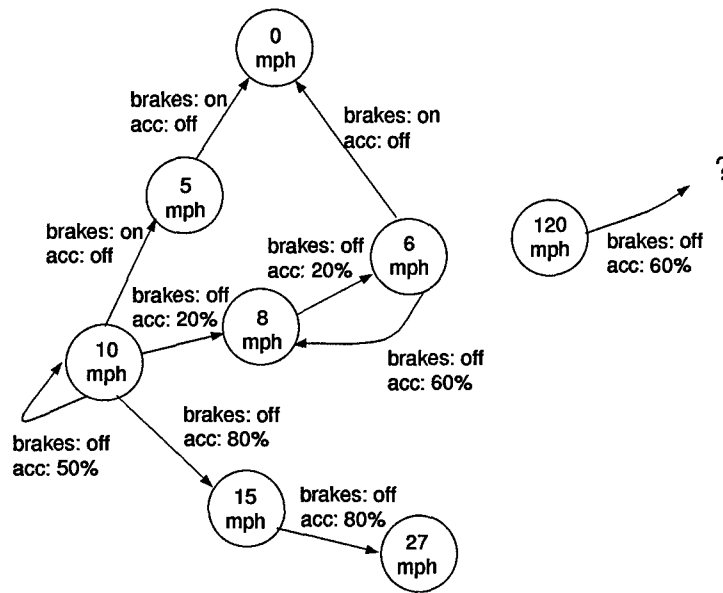


Figure 11. Hypothetical Simulation State Graph

simulation used for this research has 106 state variables, with an average of five states per state variable, giving 5^{106} (1.2×10^{74}) possible states for the simulation. It is important to note that the actual number of states the simulation can enter will likely be much smaller than this, because many, if not most, of these nodes represent illegal states.

3.2.2.1 The Meta-Control Vector. A simulation without some means of external control is of little more use than a real-world system for training. One of the main advantages of a simulation is its flexibility; the meta-control vector allows this flexibility to be exploited. For the purposes of this research, the meta-control vector allows the simulation to make transitions that would not normally occur through the transition graph. The main control requirement for the simulation is that it have the capability to be started at any state; this mechanism is essentially what allows the simulation to be used for training by giving it the means to load scenarios. This includes loading scenarios from the library for training purposes, or injecting fault states into the simulation (which couldn't be reached by any other means). A second requirement is the ability to halt the simulation or to have it run for a pre-set number of time increments. Without this capability, the simulation would essentially be unusable because it could never be stopped to present an example, or study the results of some action.

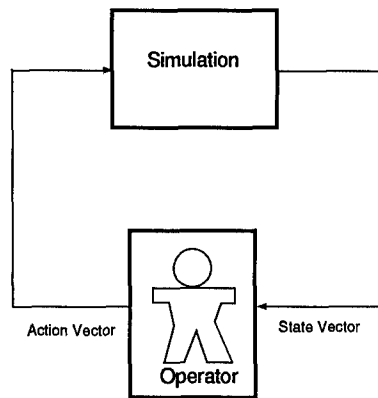


Figure 12. An Abstract Knowledge Base

3.2.3 The Domain Expert Knowledge Base. Figure 12 represents the closure of the simulation model in Figure 10; the simulation inputs actions and outputs states, while the operator does the reverse. The domain knowledge base models the operator's function, responding with the appropriate action vector for any given state in the simulation. The knowledge base is only modeled at the interface level; the internal form of the knowledge base could take any form, such as a conventional expert system, a lookup table, a specialized program, or even a neural net. However, at the *interface* level, the knowledge base acts as if it is implemented internally as a huge table of state-action pairs, with one entry for each possible state of the simulation within the span of knowledge encompassed in the knowledge base. This implies that the simulation is deterministic. Determinism is a requirement because rule induction depends upon well-structured examples.

3.2.4 Knowledge Acquisition Module. The knowledge acquisition module (KAM) is the heart of my architecture, and is the primary element responsible for making the generic approach work. The KAM feeds learned and discovered knowledge to three components of the architecture: the scenario library, the expert module and the tutor module. This knowledge comes from the KAM observing and controlling the interaction of the knowledge base with the simulation; the details of this process will be discussed in Chapter 4. To accomplish this task, the KAM must be interfaced with both the knowledge base and the simulation. The particular details of the interface will be implementation-dependent, but could take a variety of forms such as Dynamic Data Exchange (DDE), On-Line Linking

and Embedding (OLE), sockets, files, or some custom program. In particular, the KAM requires the capability to query the knowledge base based upon an arbitrary simulation state. This query ability must be reasonably quick (significantly faster than the one simulation time interval) because the knowledge query is required for every simulation cycle (timing is discussed in Chapter 5). Next, the KAM must be able to control the simulation by sending actions and receiving state information. Additionally, the KAM needs to be able to control the simulation and load it with an arbitrary scenario.

3.2.5 Control Module. The control module is the supervisor for the various ITS modules during the training phase. The control module governs the structure of the training by integrating a variety of information from the different modules to dynamically determine the next training action. First, the control module consults the student model to determine an area of training that the student needs, and that is close to knowledge he already has demonstrated. Next, the tutor module is consulted to determine the appropriate training event and instructional progression for the material. Based upon the training event, the control module may present a scenario from the scenario library to the student (through the simulation interface), or may present some other instructional information (from the expert module) through the SCIUS system interface. It is worth noting that the previous discussion is in reference to a control module *design*; the actual control module would be the last thing to be implemented in a full implementation of SCIUS. Details about the implemented elements of SCIUS are presented in Chapter 5.

3.2.6 Tutor Module. The generic tutor module embodies domain-independent pedagogical information; the domain-specific pedagogy information is acquired from the KAM. The primary purpose of the tutor module is to provide a training structure; this information is based upon Gagne's (17) instructional events. The instantiation of the proper training strategy is accomplished through the use of automatically acquired knowledge about concept complexity and relationships. For example, one part of the pedagogical strategy is to teach simpler concepts before more complex concepts. An instantiation of this concept in the driving domain might be to teach straight parking before parallel parking.

3.2.7 Student Model. The student model is based upon the classic "overlay" model, in which the student's knowledge is modeled as a subset of the expert's knowledge (54). The overlay model is certainly not the most comprehensive model, but it is a well established and adequate model. The subject of student modeling is currently a source of great controversy in the ITS community, with many researchers questioning whether modeling should be performed at all (12). Since the focus of this research is not modeling, it seemed appropriate to select the most well-established modeling approach for this architecture. In any case, the model presents a clear mapping between the student's approximate knowledge and an ideal; when the student has covered all the knowledge areas represented in the expert model, he can be considered qualified, or equivalent to the expert.

3.2.8 Expert Module. As discussed, this architecture does not contain any domain-specific knowledge; the training knowledge is acquired through the KAM. However, Figure 9 shows a block of domain-independent expert knowledge (a term that is almost an oxymoron). In this case it refers to the general structure of the expert knowledge, which is expected to be in the form of generalized state-action pairs. This knowledge is extracted from the knowledge base by the KAM through the process of induction over the scenarios discovered during scenario exploration. The knowledge in the expert module constitutes the bulk of the training material presented to the student.

3.2.9 Scenario Library. The purpose of the scenario library is to store and structure the scenarios discovered during the scenario exploration process. The library also includes the scenarios pre-specified as emergency scenarios that are not operator-induced. The scenario library is structured in terms of related scenarios that require the same action; these scenarios are all considered instances of each unique action type. These scenarios are read by the control module for presentation to the student, either as demonstrations or tests.

3.3 Summary

This chapter has presented an architecture for a domain-independent intelligent training system that requires only domain-specific knowledge in the form of a simulation and

knowledge base. The domain-specific knowledge is specified only at the interface level, so the internal representations of this knowledge can take any form. The remainder of the domain-specific knowledge is extracted automatically through scenario state-space exploration and machine learning (described in Chapter 4). The SCIUS architecture uses the standard ITS model and components, with the addition of a scenario library and a knowledge acquisition module. Furthermore, the architecture integrates unobtrusively with the operator's standard work environment, providing most of the interaction with the operator through the simulation interface, which closely resembles the real-world system interface, therefore greatly reducing the difficulty of the transition from training to the real world.

IV. Knowledge Acquisition

The previous chapter presented a framework for a generic intelligent training system that uses a simulation and knowledge base only at the interface level, without any access to the internal representations of these two components. Simply *using* these two components is not sufficient for training, because all such a system could do would be to present scenarios with their appropriate actions, with no structure to the presentation, or any explanation about *why* actions were appropriate. This chapter deals with the methods I developed to learn and extract knowledge from these two components to provide the training knowledge required to effectively utilize these components. First, justifications for why the architecture only requires external access to these components will be presented, followed by a discussion of how the internal knowledge is accessed at the external interfaces.

4.1 Accessing Internal Representations

The internal symbolic representations of the knowledge base and simulation were accessible in the original design of this research, with the idea that the knowledge contained in these components could be parsed, analyzed, reasoned about, and structured for presentation to the student. However, preliminary experiments and analysis quickly made clear the idea that the internal representations of these components are much too variable to be used for any sort of structured input to the ITS.

4.1.1 Knowledge Base Internals. Originally, the plan for this research was to utilize a conventional expert system, and parse the actual knowledge base as part of the knowledge extraction process. This approach seemed intuitive, because it would allow complete access to the antecedents and consequents of rules. Thus, there would be an easy method to explain the justification for the rules, by way of the antecedents. For example, consider the following rule in the driving domain:

```
(defrule car-ahead-brake
"Determine if there is a car close ahead of our car, and closing too
fast"
(path-ahead unclear) ;; there is an obstacle ahead
(obstacle-ahead-type car) ;; of type "car"
```

```

(distance-to-obstacle ?o-distance)
(closure-rate-to-obstacle ?c-rate) ;;
(test (< ?o-distance 50)) ;; the object is 50 feet ahead
(test (> c-rate 10)) ;; closing at 10 mph
=>
(assert (action apply-brakes 100))) ;; apply the brakes at 100%

```

If the operator gets into this situation, the system would notice this rule has fired. It is desirable to provide a good explanation of why the operator should apply this rule. "Parsing" rules could be written to provide a natural language explanation capability like:

```

(defrule object-ahead-phrase
"convert an object-ahead rule to an English phrase"
(antecedent (path-ahead unclear)) ;; find a rule that deals with an obstacle
(antecedent (obstacle-ahead-type ?otype)) ;; find the type of object
=>
(assert (phrase "There is a <?otype> in the road ahead."))) ;; gen the phrase

```

Other rules would fill out the rest of the explanation based upon the characteristics of the rule. So far, this seems like a pretty good solution; a little knowledge engineering is required to write the parsing rules, but this provides a fairly easy method to convert the expert system knowledge into a form more amenable for training. The important features of the simulation state vector are clearly identifiable in the antecedent of the rule. Presumably, anyone generally familiar with the domain could examine the rule base and create the explanation rules. However, the driving rule presented above is in an ideal format for explanation. Consider the following equivalent set of rules:

```

(defrule r001
(p-a unc)
(p ?p-state)
(equal ?p-state v001)
=>
(assert (phase d-obs)))

(defrule r002
(?phase <= (phase d-obs)) ;; This finds a pointer for a phase-change fact
(p ?p-state)
(equal ?p-state v001)
=>
(assert (obs car))
(retract ?phase)) ;; this retracts the phase-change fact to allow a change

```

```

(defrule r003
(obs car)
(d-o ?d)
(c-o ?c)
(test (< ?d 50))
(test (> ?c 100))
=>
(assert (phase wrn-crsh))))

(defrule r004
(?p <= (phase wrn-crsh))
(obs car)
(d-o ?d)
=>
(assert (action apply-brakes 100)))

```

These four rules are *functionally* equivalent to the single rule presented above, and might be the result of a particularly naive knowledge engineer. Anyone examining these rules (including the knowledge engineer after a few days) would not understand exactly what the symbols in these rules mean, nor what they are accomplishing. It is clear from this simple example that any knowledge extraction technique that relies upon accessing the internal representation of a knowledge base is going to be subject to the limitations of the knowledge engineer who created the knowledge base. This in itself is justification for why access to the internals of the knowledge base is a bad idea. However, there is an additional limitation of this technique – it is strongly dependent upon the particular expert system shell used to create the knowledge base. The rules shown above are in the C Language Integrated Production System (CLIPS) format, and CLIPS is particularly amenable to interfacing with external programs. Most expert system shells have a proprietary rule format and syntax, which implies that a custom interface would need to be built for each potential expert system shell used by the ITS. Furthermore, many expert system shells do not even have a rule output format that could be read by an external program, making interfacing with these shells impossible.

Because of the limitations described above, it was determined that a generic system could not depend upon *any* internal representation of the knowledge base. Thus, I concluded that the knowledge base would have to be accessed at the next higher level; i.e., the interface. The SCIUS architecture defines how the expert behaves at the external interface

level; this is behavior that can be depended upon, and is not subject to the variations of engineered rules. Using the expert knowledge at the interface level solves the problems described above, because it is completely independent of the internal representation. This has an added benefit, namely that because we are only using the interface level knowledge, the knowledge does not even have to be represented by forms other than an expert system. Theoretically, the knowledge could take any form that meets the interface requirements, such as a look-up table, custom program, or even a neural net. However, the interface approach does cause some new challenges. The original approach called for accessing the antecedents of the rules to isolate the important features of the rule; this is important information because it tells which features of the state vector are relevant and which are not. Later sections will discuss this problem in depth, along with the approach for solving it.

4.1.2 Simulation Internals. The previous section discussed why an ITS cannot extract knowledge from the internal representation of the knowledge base; the primary reason is the variability of knowledge base structures and platforms. No predictable structure can be expected from an arbitrarily structured knowledge base. The same argument applies even more persuasively to simulations. Knowledge bases, at least on some level, define rules and facts, so there is a relatively high of commonality among knowledge bases created under different shells. However, simulations can be created in virtually any language, having virtually any structure. The same simulation could be implemented in C, FORTRAN, Lisp, or Ada, and every simulation could have a completely different internal structure. The only conceivable method for accessing the internal structure of a simulation would be to specify a particular representation, syntax, and language for the simulation implementation, which would significantly limit the flexibility of the ITS. Most importantly, this would nullify any potential knowledge engineering gains that could be made by using an externally developed simulation.

Because the simulation internals are less accessible than the knowledge base internals, it makes sense to access the simulation only at an external interface. This approach has many of the same advantages as the external knowledge base approach. Primarily, the sim-

ulation can take any internal form as long as it meets the external interface requirements, providing significant flexibility and adaptability to a wide variety of simulations.

4.1.3 Summary. Accessing the simulation and knowledge base internals seems the most logical first step for automated knowledge acquisition because the knowledge is explicitly represented in a machine-readable form. However, variations in the implementation of this knowledge make it essentially unusable in a symbolic form; simply because a system behaves correctly does not mean that the internal representation is usable. Since the system behaves correctly at the external level, it makes more sense to access knowledge at the external level. This can be accomplished by using machine learning to acquire the knowledge required for training.

4.2 Machine Learning

Exactly what constitutes learning is a topic of debate in the machine learning community; some researchers feel chunking¹ does not qualify as learning because the knowledge obtained by chunking is already present in the system, but simply has not been realized. For the purposes of this research, I will define machine learning as any activity that creates new information not represented (or accessible) *explicitly* anywhere in the system. Because this is not a machine learning research effort, the philosophical debate as to the true characteristics of learning is not important.

The following sections will first discuss the machine learning problem, followed by my particular approach for acquiring the knowledge required for training.

4.2.1 An Illustration of The Problem. This section presents a very simple example of the knowledge base/simulation interaction. The intent of this example is to present a concrete example of the learning problem, to allow for a clearer understanding of the more theoretical discussions presented later.

¹Basically, chunking connects inference chains. For example $A \Rightarrow B$ and $B \Rightarrow C$ can be chunked to form $A \Rightarrow C$.

Consider a small fragment of the domain of automobile driving that involves making a right turn at a stop sign. A simulation in the driving domain might have (at least) the following state attributes:

1. car speed
2. car direction
3. car location
4. brake position
5. accelerator position
6. steering position
7. fuel level
8. engine rpm
9. engine temperature
10. next road sign
11. next intersection
12. next intersection state
13. next street on route

A "start vector" for a right turn could be:

car				position			engine		next path			route
#	spd	dir	loc	brake	acc	steer	fuel	temp	sign	intersection	state	street
1	22	N	Main(2.5)	off	20%	0°	9	185	Stop(.7)	Main&Maple(.8)	busy	Maple

This state vector indicates that the automobile is traveling North at 22 mph, 2.5 miles from some origin on Main street. The gas pedal is engaged 20%, and the steering wheel is at the neutral position. The automobile has 9 gallons of gas, and the engine temperature is 185 degrees. The next road sign visible is a stop sign, and the automobile is approaching the intersection of Main and Maple, which is 0.8 miles away. Finally, according to some pre-planned route, the automobile needs to be on Maple street next.

A nominal progression through states could be:

#	spd	dir	loc	brake	acc	steer	fuel	temp	sign	intersection	state	street
2	20	N	Main(2.8)	off	20%	0°	9	186	Stop(.4)	Main&Maple(.8)	busy	Maple
<action: brake = on>												
3	10	N	Main(3.0)	on	0%	0°	9	183	Stop(.2)	Main&Maple(.3)	busy	Maple
4	5	N	Main(3.1)	on	0%	0°	9	182	Stop(.1)	Main&Maple(.2)	clear	Maple
5	0	N	Main(3.15)	on	0%	0°	8.99	181	Stop(.05)	Main&Maple(.15)	busy	Maple
6	0	N	Main(3.15)	on	0%	0°	8.99	181	Stop(.05)	Main&Maple(.15)	busy	Maple
7	0	N	Main(3.15)	on	0%	0°	8.99	180	Stop(.05)	Main&Maple(.15)	clear	Maple
<action: brake = off, acc = 20%, steering = +45°												
8	5	E	Maple(.34)	off	20%	45°	8.98	198	Yield(7.0)	Maple&Elm(.7)	busy	Maple

Clearly, this is a lot of information to deal with; simulation state vectors are typically not very human-readable. In English, the automobile approaches the intersection (1-2), applies brakes until the automobile comes to a stop (3-5), waits for the intersection to clear (6-7), then makes a right turn (8). The key knowledge to discover from this example is why the actions are performed after step two and step seven. In other words, "why do those states trigger actions, and not the states before them?" From a human perspective, it may be somewhat obvious that the reason to apply brakes is because the automobile is approaching a stop sign. We could guess that the knowledge base contains some rule of the form:

```
(defrule brakes-1
(next_road_sign stop) ;; next sign ahead is stop
(distance next_road_sign ?d)
(test (< ?d .5)) ;; the sign is less than .5 miles away
=>
(assert (action brakes level on)) ;; apply the brakes
```

A more complete rule might be of the form:

```
(defrule brakes-2
(next_road_sign stop) ;; next sign ahead is stop
(distance next_road_sign ?d)
(test (< ?d .5)) ;; sign is less than .5 miles away
(desired_street ?dstreet)
(next-intersection ?street1 ?street2)
(equal ?street2 ?dstreet) ;; the street we want is at the next intersection
=>
(assert (action brakes level on)) ;; apply the brakes
```

This rule encapsulates deeper knowledge about the domain because it includes the information about the upcoming intersection containing a street that is on our pre-planned route. However, to accomplish the operator's job in this case, the extra information is actually extraneous; it is enough for the operator to realize he must stop because of the upcoming stop sign. It is important to note that to extract these two rules from the set of state vectors described above requires a significant amount of human common-sense and background knowledge. An equally valid rule which describes the state vectors could be:

```
(defrule brakes-3
(engine_temp ?temp)
(test (equal ?temp 186)) ;; the engine temperature is 186 degrees
=>
(assert (action brakes level on)) ;; apply the brakes
```

From a human perspective, this rule clearly has problems; it is correct for the given state vector example, but it does not really capture the knowledge of the domain. This rule could fail in a number of ways – it could fail to fire in a situation that required the brakes, and it could fire in an inappropriate situation that did not require the brakes. It is apparent from this simple example of rule discovery that we cannot discover the relevant state variables that trigger an action by simply looking at the variables which change.

4.2.2 Induction. The previous section presented an example of why it is very difficult to learn given no background knowledge and only a single example of a target concept. However, machine learning induction can learn with no background knowledge, by using a set of examples instead of a single example. This section describes my approach to using induction to extract the relevant features (concept) from a set of scenarios (examples).

Classic induction deals with learning a “concept” that distinguishes a set of positive examples from a set of negative examples – the concept should include all positive examples and exclude all negative examples. In this architecture, an example is a specific scenario, or state vector; positive examples are scenarios that require a specific action while negative examples are scenarios that do not require that action. A concept is a set of instantiated features that distinguish scenarios that require a specific action from other scenarios that do not require that specific action. For instance, in the driving example, there are two possible concepts that distinguish the “brakes” action: (a) (next-path-sign = stop, next-path-sign-

distance $< .5$) and (b) (engine-temp = 185). For my purposes, induction attempts to isolate a minimal set of *relevant* features that distinguish the specific action scenarios from every other scenario. This is critical information for training – without it, the student is left to flounder for himself while trying to figure out which aspects of a complex scenario are important.

The inductive algorithm used for this research is ID3 (32), a well-accepted, standard approach to induction that utilizes decision trees to construct concepts. A plethora of decision tree algorithms exist. While ID3 may not be the optimal algorithm, the use of induction in the architecture is modular, so different induction algorithms could be used in place of ID3.

Relevant scenario features are isolated using the “information theoretic” measure in ID3. Basically, a single feature is chosen that separates the examples, creating branches of a tree. Then, the branches of the tree are split on another feature, and so on, until the leaves of the tree are composed of homogeneous sets of examples. Each path from the root of the tree to a positive example leaf defines a component of a concept; the disjunction of these alternate paths defines an overall concept that distinguishes the positive and negative examples. The information theoretic measure basically splits each subtree on the feature that provides the most information. For example, the “engine-temp” feature provides maximum information because it completely splits the positive and negative examples of the brakes concept, with (engine-temp = 185). The “desired street” feature, on the other hand, provides minimal information because it does not split any examples.

4.2.2.1 An Example of Induction. Induction takes a set of positive and negative examples of a target concept, and attempts to determine the set of features that distinguish the positives from the negatives. In this example, the target concept is “brake = on,” and we would need a number of examples (state vectors) that led to this action, as well as other examples which led to no action or other actions. The inductive approach uses simple ID3 (32), which uses an information theoretic measure to determine the best features to use for classifying the examples. Basically, features are chosen on the basis of

how well they separate the examples. Consider the very small test set of examples below:

cls	spd	dir	loc	brake	acc	steer	fuel	temp	sign	intersection	state	street
+	20	N	Main(4.3)	off	20%	0°	9	182	Stop(.4)	Main&Ash(.8)	busy	Ash
+	18	N	Main(2.8)	off	20%	0°	9	186	Stop(.4)	Main&Maple(.3)	busy	Maple
-	19	N	Main(3.0)	on	0%	0°	9	181	Stop(.9)	Main&Maple(.2)	busy	Maple
-	0	N	Main(3.15)	off	20%	0°	8.99	185	Stop(.05)	Main&Maple(.15)	busy	Maple

The first two cases are positive examples of when the brakes should be applied, the second two are examples of when the brakes should not be applied. The one feature that completely partitions the examples is "Stop{.4mi}," leading to the induction of a rule similar to **brakes-1** above. The idea is that with enough examples, the non-causally related features will vary (as temperature does above), and provide less information than the causally-related features. In reality, many more examples are required for successful induction, but we have a very large pool of examples at our disposal through the scenario exploration process (described in section 4.3). In the example above, the exploration process would involve such things as applying the brakes at step one, not applying brakes at step two, and applying a steering control movement at step four.

4.2.2.2 Induction in SCIUS. Figure 13 shows a conceptual flow diagram of the rule induction process. The input to the process is a set of state-vector trace sequences that come from the scenario exploration process (described in section 4.3). Each state vector trace consists of a sequence of state vectors and their associated actions. State vector traces do not necessarily have the same length; rather, the number of vectors in a particular trace is a function of the exploration process.

A preprocessor determines the unique actions in each sequence and "labels" the state vectors as positive and negative examples of each unique action for input to the ID3 engine. Each unique action represents a separate training set for the induction engine, and is processed in a separate cycle. The number of positive and negative examples will vary between training sets, and is limited by the number of available positive examples found in scenario exploration (an essentially unlimited number of negative examples are available).

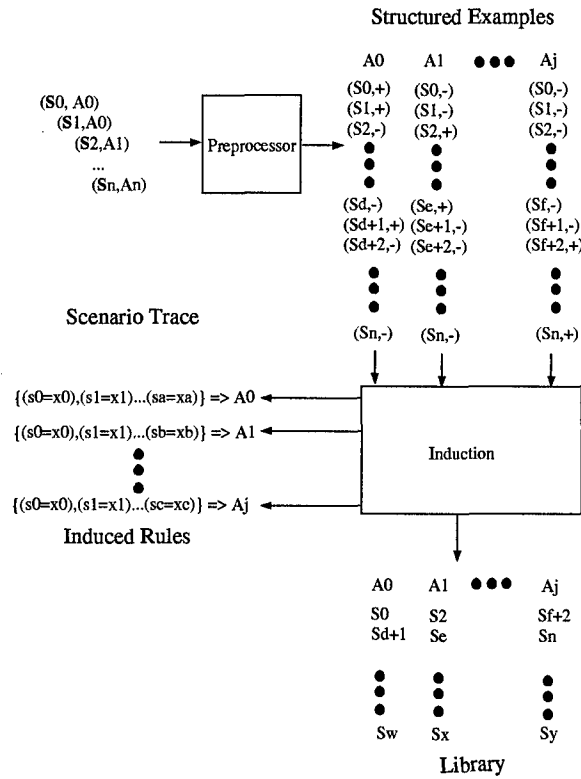


Figure 13. Rule Induction Flow

The main output of the induction engine for each unique action is an instantiated subset of the available state variables that require the action. Essentially, this is a rule consisting of an antecedent that contains the relevant state variables and a consequent that contains the appropriate action. Additionally, as part of this process, the scenarios are stored in the scenario library, grouped by unique action.

The induced rules form the core of the domain knowledge used for training; they represent an approximation of the knowledge contained in the expert knowledge base.

The induction process has the effect of converting a deep knowledge base into a shallow knowledge base, as shown in Figure 14. This induced knowledge base behaves exactly as the original knowledge base, within the scope of the original training set of scenarios. The antecedents of the induced rules provide the justification for why the action is performed while the rules themselves provide a means of determining the appropriate action to perform in any given situation. The induced rules provide the information necessary to

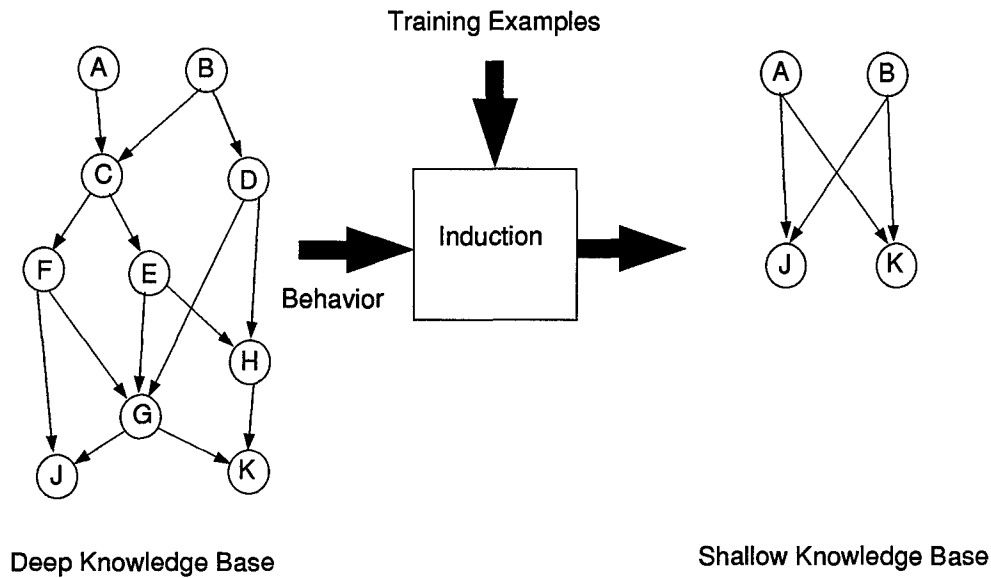


Figure 14. Knowledge Base Flattening

train using Merrill's Component Display Theory. The antecedent allows for explaining or testing a rule (expository and inquisitory general), while the induced rule itself allows for explaining or testing examples (expository and inquisitory specific).

4.2.2.3 Macro vs. Micro Action Representation. There are two possible interpretation of what constitutes an action. An action could be considered either a set of control vector inputs (a macro action) or a single control element input (a micro action). The first interpretation (macro action) will lead to more (potentially combinatoric) potential actions because every action that occurs during exploration is a unique action. For example, a macro action that consists of the control vector $[a, b, c]$ is different from the macro action $[a, b]$, which is different from the singleton macro action $[a]$.² A micro-action, on the other hand only consists of a single control input. Thus, the control vector $[a, b, c]$ is treated as three separate actions. This approach results in fewer potential actions, because micro-actions do not combine combinatorically. This may seem like a trivial distinction, but it is actually quite important to the induction process.

²But the action $[a, b]$ is equivalent to $[b, a]$. Order is unimportant.

The induction process treats a single instantiation of an action (of whatever type) as a positive example. For macro-actions, this simply means that the entire macro-action represents a positive example. Using the micro-action approach, actions are broken down into micro-actions (single control inputs), which represent multiple positive examples and therefore multiple runs of the induction process. Macro-actions, however, present a potential flaw in the induction process that prevents their use. Consider the following abstract knowledge base, where W , X & Y are states and a and b are control element inputs:

$$\begin{aligned} W \wedge X \wedge Y &\Rightarrow [a, b] \\ W \wedge Y &\Rightarrow [a] \\ W \wedge X &\Rightarrow [b] \end{aligned}$$

If an example that leads to action $[a, b]$ occurs, then the induction process will not be able to complete. Using macro-actions, $[a, b]$ will be treated as a positive example, then $[a]$, then $[b]$, all on separate runs of ID3. While $[a, b]$ is being treated as a positive example, $[a]$ and $[b]$ will be considered negative examples. However, if induction uses $W \wedge X$ as part of a concept for $[a, b]$, then there will be no remaining features to split the decision tree to exclude $[b]$. Induction will not be able to find a concept to completely split the positive and negative examples, unless there happens to be some other unrelated feature that serves that purpose.

However, using micro-actions, the following rules will be induced:

$$\begin{aligned} W \wedge Y &\Rightarrow [a] \\ W \wedge X &\Rightarrow [b] \end{aligned}$$

These rules are functionally equivalent to the knowledge base presented above.

4.2.2.4 Induced Rule Verification. One interesting (and unexpected) side-effect of the induction process is the pseudo-verification of the knowledge base that happens when the rule is induced against the context of the simulation. It might seem that the ideal situation would occur when the induced rules match the actual rules exactly. However, as previously described, the knowledge base could be implemented quite poorly, and still function correctly at the interface level. Verification is a software engineering process

that deals with whether a system has been built correctly at the syntactic level. This is contrasted with validation, which deals with whether a system has been designed correctly to solve the intended problem. Validation requires knowledge of the domain, while verification does not. The remainder of this section will discuss the eight possible syntactic errors presented in (19), and describe how the rule induction process addresses these errors in a knowledge base. In effect, the induction process discovers a “streamlined” version of the knowledge base by *not* discovering some rules that have syntactic errors.

Redundant rules Redundant rules can have syntactic or semantic redundancy. Syntactic redundancy means the rules have identical antecedents and consequents, although not necessarily in the same order. For example, $A \wedge B \rightarrow C$ and $B \wedge A \rightarrow C$. If a knowledge base contained both of these rules, the rule induction process would essentially eliminate one of them because the induction process recognizes that the order of the antecedents is irrelevant. When the induction process arbitrarily found one of the rules first, and then discovered the second rule, it would recognize that the rules are equivalent and would not add the second rule to the induced knowledge base.

Semantic redundancy, on the other hand, involves two rules that are identical semantically. For example, $\{A \wedge B \rightarrow \text{thunderstorms}\}$ and $\{B \wedge A \rightarrow \text{electrical storms}\}$. Because the induction process is a domain-independent process, there is no way for this type of redundancy to be uncovered. However, it is likely that this type of redundancy would be uncovered in the process of interfacing the knowledge base to the simulation, because there would be two actions with different symbols that accomplished the same task.

Conflicting Rules Conflicting rules are rules that have identical antecedents, but opposite consequents, such as $A \wedge B \rightarrow C$ and $A \wedge B \rightarrow \sim C$. The result of this error under induction would depend upon the conflict resolution strategy and monotonicity of the inference engine. If conflicting facts are allowed, then induction will not be able to find a concept to separate positive and negative examples of C and $\sim C$. If only one of these is allowed, then induction will find the concept that covers the fact that is allowed. In either case, induction will not properly avoid this type of error.

Subsumed Rules Subsumed rules are rules in which the antecedent is a superset of another rule's antecedent, such as $A \wedge B \wedge C \rightarrow D$ is subsumed by $A \wedge B \rightarrow D$. This type of error is avoided by the induction process because the induction algorithm discovers the simplest concept that covers the examples, so the more complex rule will be ignored.

Circular Rules Circular rules employ a circular reasoning process, such as $A \rightarrow C$ and $C \rightarrow A$. This results of this type of error will depend upon the particular expert system shell used; if the shell does not implement refraction, then circular rules will cause the shell to become stuck in an infinite loop. With refraction, both rules will be learned.

Dead-end Rules Dead-end rules have consequents that are not part of the antecedent of any other rule, nor are they part of the goal of the system. In other words, these rules assert facts that are useless to the operation of the knowledge base. The induction process will ignore these types of rules because their antecedents contribute no information to the splitting of positive and negative examples.

Missing Rules A missing rule occurs when there are facts that are not used by any rules. The implication is that there is some rule that should use the facts, although it could also be the case that there are just extraneous facts in the system. In any case, the induction process could be used to find "missing" rules by examining the difference between the number of state variables in the input to the knowledge base, and the number of state variables used in the induced rules.

Unreachable Rules An unreachable rule is the inverse of a dead-end rule; it is a rule that has a antecedent that can never be met under any circumstances. These rules will be ignored because they will never fire, and thus are essentially not part of the knowledge base from the interface level.

4.2.2.5 Other Effects. In addition to the formal syntactic errors describe above, there are a few additional beneficial side-effects of the induction process on the input knowledge base. These are the result of the induction process working with the context of the simulation; these effects are not corrections of syntactic errors, but are effects that streamline the knowledge base.

Chunking Chunking takes a series of inference chains and collapses them to a single rule, such as $A \rightarrow B$ and $B \rightarrow C$ will be chunked to $A \rightarrow C$. Some causal information is lost in the chunking process; the new rule represents shallower, but more efficient knowledge. The induction process performs chunking because it works only at the interface level of the knowledge base. In the example above, the induction process uses only the input of 'A' and the output of 'C.' The intermediate step of 'B' is not available information.

Causally extraneous antecedent elements One tendency of knowledge engineers is to err on the side of caution, and add antecedents to a rule to ensure that it does not fire at an inappropriate time. However, in practice some of these extraneous antecedent elements will never occur, making them superfluous. The induction process works within the context of the simulation; if a rule says $A \wedge B \wedge \sim C \rightarrow D$, and C never occurs during simulation exploration, then only $A \wedge B \rightarrow D$ will be learned.

Symmetrical Rules Symmetrical rules are rules that differ only by one (or a small number of) antecedents, like $A \wedge B \rightarrow D$ and $A \wedge C \rightarrow D$. If, during simulation exploration, 'B' and 'C' always occur together, then only one of these rules will be learned, because the other is essentially extraneous within the context of the simulation. Again, this is an efficiency trade-off. There is causal information that is lost when one of these rules is eliminated, but the remaining rule performs the job of two rules.

Completeness Although the induction algorithm does tend to streamline the original knowledge base, it does have some limitations, namely that there is no way to know whether the input knowledge base has been completely discovered. The completeness of the induced knowledge base is a function of the variety of the training examples. If a certain rule deals with a particular situation that never occurs in the entire set of training examples, then that rule will never be induced. Later sections deal with the scenario exploration process, which addresses the issue of how many scenarios are needed to fully "expose" the expert knowledge. In any case, there is not an issue of accuracy, because the system continually verifies the accuracy of the induced knowledge base against the expert knowledge base; any discrepancies are corrected by further induction.

4.2.3 Summary. The induction process takes a set of representative examples (scenarios) and determines the relevant features (state variables) that lead to the desired action. The ID3 algorithm will tend to find simpler concepts that cover the available examples, making ID3 an appropriate induction algorithm, although any alternate algorithm could be used. The output of the induction process is a streamlined, partially verified, potentially incomplete version of the original knowledge base, along with a structured set of scenarios that constitute the scenario library. The induced knowledge is the core of the domain-specific training knowledge; it allows concepts and principles to be presented and tested along with the presentation and testing of scenarios.

4.3 Scenario Exploration

The previous section described the induction process, which used a representative set of scenarios to determine the relevant features and causes of particular actions. The mechanism for finding the scenarios is the scenario exploration process; this process searches the scenario state space in a systematic manner to find a representative set of scenarios in a infinite, or extremely large state space. Additionally, the scenario exploration process finds scenarios that are used for training (via the scenario library).

4.3.1 The Problem. The crux of the problem is how to find a finite set of training examples that covers an extremely large state space, with no domain knowledge for guidance. Typically, scenarios are crafted by hand to optimally illustrate certain aspects of training, but this process is knowledge and labor intensive and provides only a relatively small, static set of scenarios. Additionally, hand-crafted scenarios must be updated by hand whenever knowledge changes. In contrast, an automated scenario exploration process allows an essentially unlimited supply of new scenarios, and can automatically generate new scenarios whenever knowledge changes.

One obvious alternative for scenario generation is sampling, which I call state-based exploration. This technique involves defining a range (or set) of possible values for each variable, and using this information to generate random state vectors. The advantage of this approach is that it is statistically likely to generate a good covering of the scenario

state space. The generated vectors will likely be representative of the state space as a whole. However, this approach has one fundamental flaw – there is no way to determine whether a particular state is “legal” or not. An illegal state is defined as a state for which the simulation was not designed; Figure 11 in Chapter 3 presented an example of an illegal state. At first, it was thought that this problem could be overcome by using the simulation as a sort of “filter,” with the idea that the simulation would self-correct by transitioning towards legal states. As objects in the simulation influenced each other, they would force illegal states back to legal states. I also believed that there would be many more legal states than illegal states. Both of these arguments turned out to be flawed.

First, if a simulation is forced into a state that it is not designed for, then transitions from this state are undefined. The simulation could crash, never transition, or transition to a completely unexpected state. A robust simulation could be designed to correct from illegal states, or filter out illegal states from the scenario input process, but there is no justification for believing an arbitrary simulation would work this way or have these capabilities. Second, there may be an infinite number of illegal states. Consider a simulation of a steam boiler, where pressure and temperature are directly related. The legal states of pressure and temperature define a line; every pressure-temperature combination outside this line is illegal. In this situation there are an uncountably infinite number of illegal (and legal) states.

The fundamental problem of state-based exploration is the presence of a potentially infinite number of illegal states, making sampling techniques infeasible. Without some means to discriminate between illegal and legal states, illegal states could be used for either concept induction or student training, both of which would have negative effects upon overall training correctness and effectiveness. It is not reasonable to expect a simulation to have this discrimination capability, so some other means of discrimination must be employed to ensure only legal scenarios are generated.

4.4 Action-Based Exploration

The simulation is not guaranteed to do anything when it is forced into an illegal state. However, the simulation *is* guaranteed, by definition, to only transition *to* legal states *from*

legal states. Actions (including the null action) cause only legal transitions; thus, by sampling the *action space* of the simulation instead of the state space, an action-based exploration process will find only legal states.

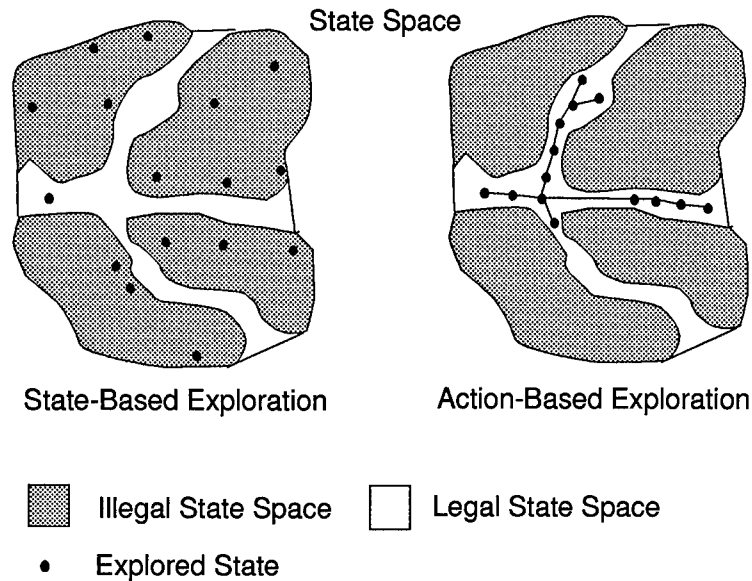


Figure 15. State vs. Action Based Exploration

One disadvantage of the action-based approach is that the explored scenarios are not necessarily representative of the simulation state space as a whole – they do not necessarily provide a good covering of the space. This is because the action-based process is essentially sequential; it explores connected sequences of nodes through the space, whereas a state-based approach “jumps” around the space. An action-based approach may eventually find a representative set of nodes in the state space, but given the same number of nodes, the output of the action-based process will be less representative than a state-based process. This difference in covering is shown conceptually in Figure 15. In a strongly-connected³ simulation state graph, the entire legal state space can be found from a *single* legal node. However, in reality most simulations are not strongly-connected. The implications of this will be discussed below.

³A strongly-connected graph is a graph where any two arbitrary nodes are connected by some path.

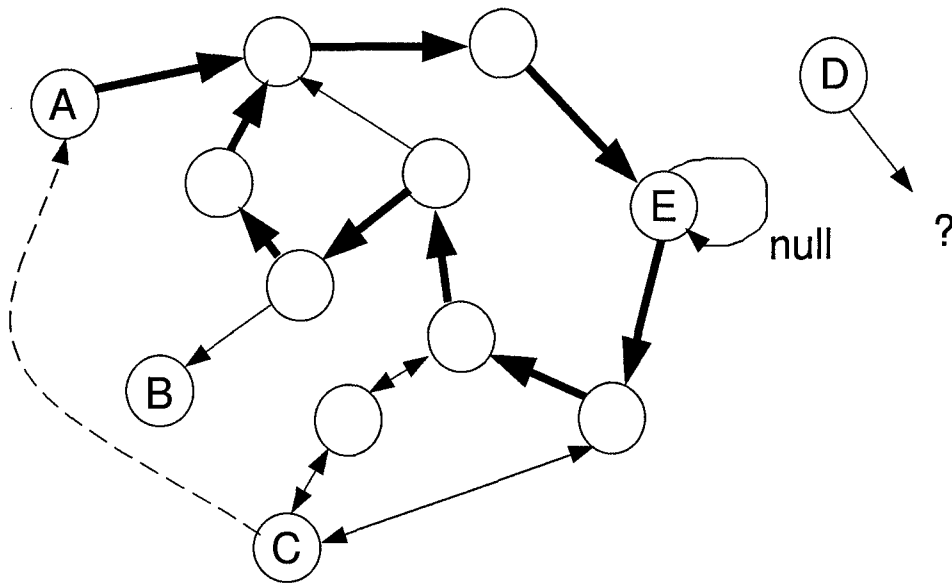


Figure 16. Example Simulation Graph

4.4.1 Simulation Graphs. Figure 16 shows a fragment of a prototypical simulation graph, illustrating some of the problems with a non-strongly connected graph. Node A represents a source node; any node in the graph (except D) can be reached from node A. Node B represents a sink node; once the simulation reaches this state, it makes no further transitions. Node D is an illegal node, transitions from this node are undefined. Node C represents a corrected deviation from the expert path (the darkened arcs). Node E is a steady-state node; the null action will keep the simulation in that state indefinitely. Finally, the transition from node C to A represents an unnatural transition, one that could not occur through some operator action, but is instead forced by using the simulation meta-control vector.

An example of a source node might be the start-up state of a nuclear reactor; it might be the case that the simulation is not designed to handle shutting down the reactor, but it is desired to simulate the start-up process. Another example of source node is a component failure. Most training simulations will not model systems at the component level, with component reliability models, failure modes, etc., because failures would happen too infrequently to be used for training. Instead, the simulation will be "forced" into a component failure state for the purposes of training; this failure state is a source node

because it cannot be naturally reached by any other state. Source nodes are problematic because they can never be found through a domain-independent exploration process. For the scenario exploration process to work, all source nodes that are important to the simulation must be specified in advance, either through a state vector description, or through some accessible procedure call. It is reasonable to expect that this information is available in an independently-built simulation, because without it, the simulation wouldn't be particularly usable because it could never reach these states.

A second problem with non-strongly-connected simulation graphs is sink nodes such as node B in the figure. A sink node represents a "catastrophe" situation; i.e. the simulation designer felt it was not worth simulating anything beyond that point. Examples of sink nodes might be a plane crash in an aircraft simulation or a meltdown in a nuclear power plant simulation. Sink nodes are also problematic because a strict action-based exploration process will halt at that point, and will not discover any new nodes. This problem is detected, however, by using a "watchdog" timer that forces the simulation into a new (or previous) state whenever the exploration process appears stuck. However, the one problem with this approach is the presence of cycles in which the simulation appears to be making transitions to new states, but is in reality navigating a circular path through the state space. Cycle detection is a complex problem, beyond the scope of this research.

The methods described above eliminate the disconnectedness of source and sink nodes, allowing the simulation state graph to be treated as a strongly-connected graph. With a strongly-connected graph, the exploration process can potentially find any node from any other node, so the entire state space can be explored.

The expert path in Figure 16 represents an "ideal" progression through the simulation, given some initial starting node. As a student continues to perform inappropriate actions, he deviates further and further from this ideal path. This concept is shown in Figure 17. If the student later performs an appropriate action, then he may immediately return to the expert path (1), he may have to pass through several intermediate nodes to reach the expert path (2), or he may reach an entirely different expert path (3). It is desirable to keep the student as close to the ideal path as possible; as he progresses farther and farther from the ideal path, he may reach a state which prevents access to the original

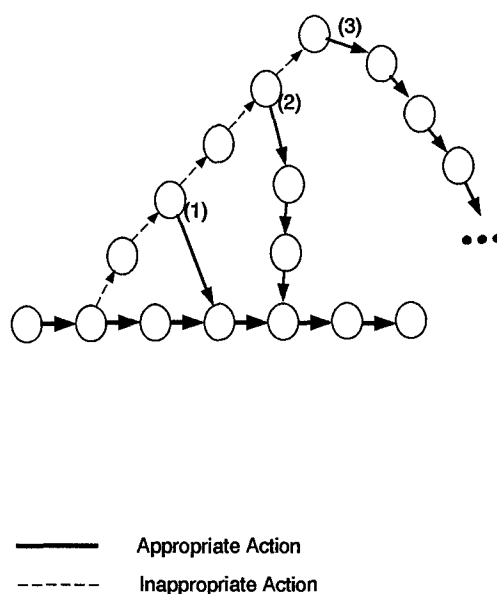


Figure 17. Expert Path Deviations

ideal path. For example, if an overheating reactor is ignored long enough, it may reach a state at which cooling it to the ideal temperature is impossible.

4.4.2 The Exploration Process. The overall goal of the exploration process is twofold: to “expose” as much of the expert knowledge as possible and to provide a set of appropriate scenarios that can be used for training. Expert knowledge is exposed when a particular state causes a rule to fire (or some equivalent knowledge component in another knowledge representation), suggesting an action to perform. The second goal is accomplished by the nature of the exploration process; instead of finding a set of scenarios that is representative of the state space, the action-based process finds a set of scenarios that is representative of the situations the operator is likely to encounter.

As previously discussed, a state-based exploration process is statistically likely to provide a good covering of the simulation space, whereas an action-based process will tend not to provide a good covering of the entire space. However, a representative covering of the simulation space may not necessarily be desirable, because the simulation space encompasses a much wider variety of states than the student will ever face as an operator. At any given simulation state, there is a single appropriate action (control vector) to perform, and a much greater number of inappropriate actions. As a student continues to

perform inappropriate actions, he potentially reaches an exponentially growing number of simulation states that are removed from the expert path. Because the number of "deviant" simulation states grows exponentially, the number of potential training concepts also grows exponentially. For example, if there are 100 possible unique actions at any given state, then there are potentially 100 unique states reachable from the original state in a single step. However, in two steps, there are potentially 100^2 reachable states, and so on. In reality, many of these states will not be unique, but the potential is there – the farther the student is removed from the expert path, the more concepts we need to teach in order for him to return to the expert path, if it is even possible.

The combinatoric explosion in deviant path concepts further reinforces the idea that we want to keep the student as close to the ideal path as possible. Instead of teaching a state that is reached by multiple inappropriate actions, and has an equally complex return path, it is better to prevent the student from reaching that state in the first place. Because of this, a good covering of the simulation space may actually be a liability because it will contain more states that are far removed from the expert path than states close to the expert path. Given limited resources, it is better to concentrate on the situations that the student is most *likely* to face, than to concentrate on situations that are representative of the entire simulation space, but unlikely to be encountered.

Keeping the student close to the expert path provides a constraint that allows the search of an infinite space to be feasible, and to provide meaningful results. This is accomplished through a technique called "iterative spreading." Basically, the process works by first finding the expert path through the simulation for a given starting point (typically the source nodes), providing a baseline stream of states. The expert path is found by setting the simulation to the initial state, then querying the knowledge base to find out what action is recommended. This action is then executed, causing the simulation to change state, and then the process repeats.

Next, the expert path is followed again, but at each state a deviation (an inappropriate action) is injected. Deviations come from sampling the action space, which is a pre-specified list of actions with possible instantiations of those ranges. An example of this might be: (brakes{on,off}, accelerator{0%-100%},turn-signal{left,right,neutral}).

This information represents additional knowledge that must be input to the system, but this knowledge should be readily available as part of the interface between the simulation and the knowledge base.

After a one step deviation path is explored from the expert path, a two step path is explored, and so on, providing a gradual spreading exploration path from the initial expert path. This process is iterative, because at each step the explored scenarios are used for induction, providing a new set of discovered concepts. The process halts when no new concepts are discovered during a particular exploration set, suggesting that the expert knowledge has been reasonably exposed. This is clearly a heuristic measure, but there are no detrimental consequences if knowledge is not exposed, because of the consistency checking process, described in Section 4.5. Basically, if any knowledge is not discovered during the exploration process, but is encountered during the normal training operation of the system, it will be integrated into the available knowledge.

4.4.2.1 Steady State Nodes. One problem with the exploration technique is nodes such as node E in Figure 16. These "steady-state" nodes are states where no action is appropriate; the simulation/knowledge base combination typically stalls at these points because no action is required, so no state change takes place. At first, it would seem that this type of condition could be detected by the exploration process because the simulation would be stalled without a state change for an extended period of time. One technique for addressing this problem would be to use a type of watchdog timer; if the simulation did not change state after some number of time intervals, then the simulation could be forced to some other state or source node. However, one possible problem with this technique is a "delayed state-change" node. This type of node occurs when an action starts a timer for some process that is hidden in terms of the state vector description, essentially "behind the scenes." For example, a nuclear reactor simulation might require that the core be heated before the reactor is started. If the core temperature has two possible states: "cold" and "warm," then the simulation will effectively stall while the reactor is heated. No action will be required, and no state change will be apparent. However, after some period of time, the simulation will change state without any action on the operator's part. A strict

watchdog timer might mask this sort of state change because the watchdog would force a state change before the naturally occurring state change occurred. The only certain method for dealing with this problem is to have meta-level simulation knowledge about delayed state-change nodes, and set the watchdog timer accordingly.

4.4.3 Summary. There are two types of scenario exploration, state-based and action-based. State-based exploration finds a set of scenarios that are representative of the entire simulation state space while action-based exploration finds scenarios that are representative of the scenarios an operator is most likely to face, making it more ideally suited for training. The goal of the action-based process is to provide scenarios that expose as much of the expert knowledge as possible, and to provide a representative set of training scenarios for the scenario library. The action-based process works by sampling the action space of possible operator actions, and implementing these actions causing iteratively further deviation from the ideal expert path. This process continues until no new knowledge is obtained in a particular exploration level. The output of the exploration process is a set of scenarios that are used for the training scenario library, and as input for the induction process described previously.

4.5 Consistency Checking

One unique feature of this architecture is that the entire knowledge acquisition process does not have to be complete. Classic inductive learning research is concerned with learning a concept from a set of training examples that can correctly classify unseen examples. Inductive learning researchers typically do not have access to a "perfect classifier," (called an oracle) that can unfailingly classify any possible example, because it would defeat the purpose of induction. However, my use of inductive learning is somewhat different from the classic research approach. The knowledge base acts as an oracle; it can, by definition, correctly classify any possible example (scenario). Inductive learning is used to isolate the relevant features that cause an action, not to develop a concept that can be used independently of the oracle. Because of this, I am less concerned with the completeness of the induced concepts. The induced concepts will be correct within the scope of all examples

that have been encountered; if the exploration process works well, than the concepts will be nearly complete. However, it could certainly be the case that some unusual scenario is encountered that is not properly covered by the induced concepts. This inconsistency represents new knowledge that must be integrated into the system.

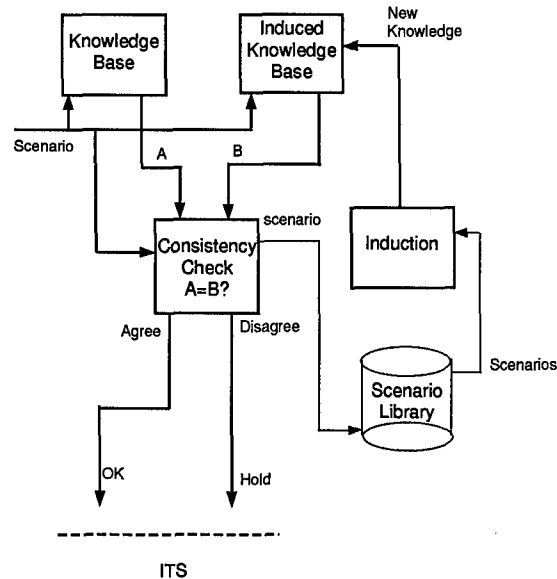


Figure 18. Consistency Checking

This process is shown conceptually in Figure 18. A scenario is the input for any type of knowledge base query. Whenever a query occurs, the output of the real knowledge base is checked against the output of the induced knowledge base; if the two agree, then the ITS proceeds with whatever task it was accomplishing. This process must be reasonably quick, faster than a single simulation time interval as discussed in Section 5.4.2. However, if they disagree, then the ITS goes into a hold state while the scenario is saved by the induction process for off-line processing. This new scenario can not be used for any explanation process, however, because it is not consistent with the induced knowledge base. It can only be presented as a shallow stimulus-response type concept without any information as to why the action is appropriate, until the new scenario is integrated into the induced knowledge base. Although the use of this scenario is limited pedagogically, this is preferable to teaching the student a concept which may be wrong.

This new scenario represents valuable information because it is inconsistent with the existing induced knowledge base, and it is a scenario that the student has at least some likelihood of facing (because it was encountered). The induction process uses this new example, along with pre-existing examples to induce a new concept that encompasses the new example. This new knowledge is then added to the induced knowledge base. In this manner, the completeness of the knowledge in the ITS increases over time; eventually it will be unlikely that scenarios will be uncovered that are in conflict with the induced knowledge base. Additionally, continual consistency checking allows the initial knowledge acquisition process to be less than perfect; any omissions will eventually be discovered by the normal training process.

4.6 Curriculum Extraction

A final aspect of knowledge acquisition is the automatic development of a baseline curriculum from the exploration of the simulation and knowledge base. The concept of a "curriculum" covers a broad range of organizational schemes, but for the purposes of this research, a curriculum is an ordering of training concepts. The overall structure of training is accomplished in line with Gagne's instructional events (17), but the instantiation of this template is accomplished by the curriculum extraction process. Basically, the extraction process uses domain-independent heuristics to structure the presentation ordering of particular concepts. Examples of domain-independent heuristics are: "teach simpler concepts before complex concepts," and "teach more frequently occurring concepts before rare concepts." Concept complexity is measured by the number and arity of preconditions in a particular induced rule. Additionally, simpler concepts may be proper subsets of more complex concepts; this represents a prerequisite for the more complex concept. The progression from simpler to more complex concepts defines a hierarchical structuring of the overall concept space. It is important to note that the curriculum extraction process develops a *baseline* curriculum; the actual curriculum will be customized during the training session by the student model and actions of the student.

The customization process is not particularly complex. The curriculum extraction process will provide a directed graph of material for the student to learn. An ideal student

will progress down the graph with no deviations. However, if a student has difficulty with a particular concept, he can cycle back to the prerequisites for that concept for remedial training. Additionally, the amount of time spent on a concept, and the specific emphasis will depend upon the particular student. For example, if a student has difficulty with one component of an action vector in a concept, then the scenarios chosen from the library will emphasize that component.

4.7 Summary

This chapter presented a description of the knowledge acquisition process, the core of the generic training system architecture. Knowledge acquisition consists of two interrelated main processes: scenario exploration and rule induction. Scenario exploration traverses the simulation state space to discover representative scenarios for training. These discovered scenarios are used by the induction process to induce a streamlined representation of the original knowledge base. Incompleteness in the induced knowledge is corrected by the consistency checking process. Finally, an additional aspect of the knowledge acquisition process is the automatic construction of a baseline curriculum, based upon the discovered knowledge and structure of explored scenarios. The previous discussion has been at a domain and implementation independent level; the next chapter will discuss the proof-of-concept system, and some implementation issues important for the actual realization of the generic architecture.

V. Implementation Issues

The purpose of this chapter is to present the results obtained by the development of the prototype system, which consists of a basic implementation of the essential aspects of SCIUS, along with a demonstration simulation and knowledge base. Along with the prototype results, various implementation-level issues will be discussed. The purpose of the prototype system is to demonstrate the SCIUS architecture in practice and to highlight interesting aspects of SCIUS's behavior. It is believed that the prototype implementation is representative of a real-world implementation of SCIUS, but there is no means for claiming statistical validity of the results presented in this chapter across training domains as a whole.¹ First an overview of the prototype will be presented, followed by detailed discussion of the simulation, knowledge base, and induction issues. Finally, a discussion of the limitations of the SCIUS model will be presented, followed by the conclusion and summary of results.

5.1 Introduction

The SCIUS prototype consists of three main components: a basic implementation of the SCIUS architecture, a demonstration knowledge base, and a demonstration simulation. There were three main priorities in the design of the SCIUS prototype:

- Build a system that is substantially more than a "toy" system.
- Build a system that involves a minimal amount of non-research related software development
- Build a system that is representative of real-world training domains.

The first priority was important because many AI systems have appeared to be successful on toy problems, then suffered difficulties when attempting to extend them to larger, more complex, realistic problems. The second priority was used because of the

¹It is worth mentioning that there is no conceivable method for obtaining statistical verification of a training architecture, because there is no way to sample training domains. Experiments could be conducted with human students to determine how much they learned versus a control group, but this would provide verification of the training model, and not the architecture used to train with the model.

nature of research, and time limitations. The third priority was considered in order to make the results of the prototype as meaningful and applicable to training as possible. The second priority is in direct conflict with the other two, but it was the overriding priority because of time and resource limitations. These priorities impinge on two elements of the prototype design: the depth of the implemented SCIUS architecture, and the choice of the prototype domain.

The depth of the SCIUS prototype architecture is sufficient to demonstrate the important aspects of this research. A full implementation of SCIUS would involve the complete operation of the system, from initial knowledge acquisition to student interaction. For the purposes of this research, it was decided to develop a prototype that implemented only knowledge acquisition. The SCIUS prototype discovers the knowledge required for a standard training model (as presented in Chapter 2), but the actual presentation and interaction with the student is not implemented. Building the remainder of the SCIUS architecture would be a nontrivial engineering task, but would not be interesting from a research perspective.

The choice of the demonstration domain for the SCIUS prototype was also motivated primarily by the second priority described above. It would seem that the easiest method for finding representative domain data would be to use real-world data. However, data from a single real-world domain may not be representative of real-world domains in general. Additionally, real-world data is typically difficult to obtain because it involves working with other organizations and unknown formats. On the other hand, a hypothetical domain, consisting of a knowledge base and simulation that do not represent a real-world domain, has several advantages. The main advantage is complete freedom in the construction of the knowledge base and simulation. Using real-world domain data will frequently be an "all or nothing" approach, but with a fictional domain the domain components can be constructed only to the level required to ensure the domain is more complex than a toy domain. Knowledge engineering is greatly simplified because the knowledge in the fictional domain is not constrained by real-world requirements²; the knowledge engineer is not required to learn the domain. This flexibility allows a variety of knowledge "idiosyncrasies" to be built

²It is instead constrained by the knowledge engineer's imagination, which is a different problem.

into the knowledge base, without concern for how well they map to the real-world. This, in turn, allows SCIUS to easily be tested on these idiosyncrasies, to learn how they affect the overall knowledge acquisition process. Finally, the choice of a fictional domain has advantages for later potential education experiments, because the subjects will not have any pre-existing knowledge of the domain. For these reasons, the SCIUS prototype uses a fictional domain.

The intent of this research was to investigate the *process* of knowledge acquisition in an intelligent training system. The design of the prototype reflects this emphasis. A real-world training situation could have been modeled, but this would have involved an extensive amount of knowledge engineering without contributing to the investigation of knowledge acquisition. A complete prototype could have been developed, and experiments conducted, but this would have only served to verify the training model used, and not the process of knowledge acquisition used.

5.2 *Prototype Architecture*

The SCIUS prototype consists of a limited implementation of the SCIUS architecture, along with a domain simulation and knowledge base. The SCIUS prototype architecture is mostly complete, with the exception of the presentation components; i.e. the SCIUS prototype discovers and configures the knowledge required for training, but does not present it to the student. The presentation component could be implemented in a generic fashion, however. As discussed previously, the instructional interface to the student should be minimal; the majority of the student interaction is conducted through the simulation interface. The instructional interface would need to present the following types of information: student directive (watch example, practice, demonstrate), student example informational (corrective feedback, concept presentation, action presentation), and curriculum informational (related-concepts, high-level concept, summary). This could be implemented in a domain-independent interface, and the required information could be presented through the interface by the presentation component of SCIUS. In a full implementation of SCIUS, the domain course designer would not need to develop the presentation interface, or provide any domain information for presentation.

The fictional domain chosen was the operation of an antimatter reactor in a fictional starship. This domain represents a prototypical CDS operator task, and involves such tasks as system reconfiguration, monitoring, and emergency handling. Much of the domain knowledge was generated from scratch, but some concepts were developed from a technical manual (48). The knowledge base is implemented as a C Language Integrated Production System (CLIPS) knowledge base consisting of 55 rules. The simulation is implemented in the Common Lisp Object System (CLOS), under Windows NT. The simulation consists of 30 objects and 106 total object attributes. The SCIUS prototype architecture is also implemented in CLOS. While the prototype system is smaller than a typical real-world system, it is more substantial than a toy system.

5.2.1 Knowledge Base. The knowledge base consists of mostly "flat" rules, involving a state-based antecedent and an action consequent. An example of such a rule is:

```
(defrule ari-stress
  "if the ari is stressed, turn down the am level"
  (ari state stressed)
  (ari level ?level)
  (test (> ?level 20))
  =>
  (assert (action ari level 20)))
```

This rule simply states that if the antimatter reactant injector (ARI) is stressed, and the level of the ARI is above 20, then set the level to 20. However, there are a number of more complex rules implemented, to test various aspects of SCIUS's operation. First, *phase* rules control the firing of different sets of rules, based upon the desired configuration of the reactor. For example, if an order is given to set the reactor to "cruise" configuration, and the reactor has not been started yet, then the knowledge base will assert phase rules to implement the start-up procedures. If the reactor has already been started, and the order for cruise is given, then the start-up rules will not fire. Additionally, there are *asymmetric* and *symmetric* rules, that deal with symmetric aspects of the simulation. Asymmetric rules differ in threshold values for symmetric simulation components while symmetric rules are

essentially the same for symmetric simulation components.³ Finally, rules with extraneous preconditions are implemented.

5.2.2 Simulation. The prototype simulation is implemented in Common Lisp, using the Common Lisp Object System (CLOS) as a means of modeling the simulation objects. Thirty simulation objects are modeled, consisting of 106 attributes; a simulation state vector is 106 elements long. The simulation runs under Windows NT, and implements a graphical user interface (GUI) to allow the operator to control the simulation.

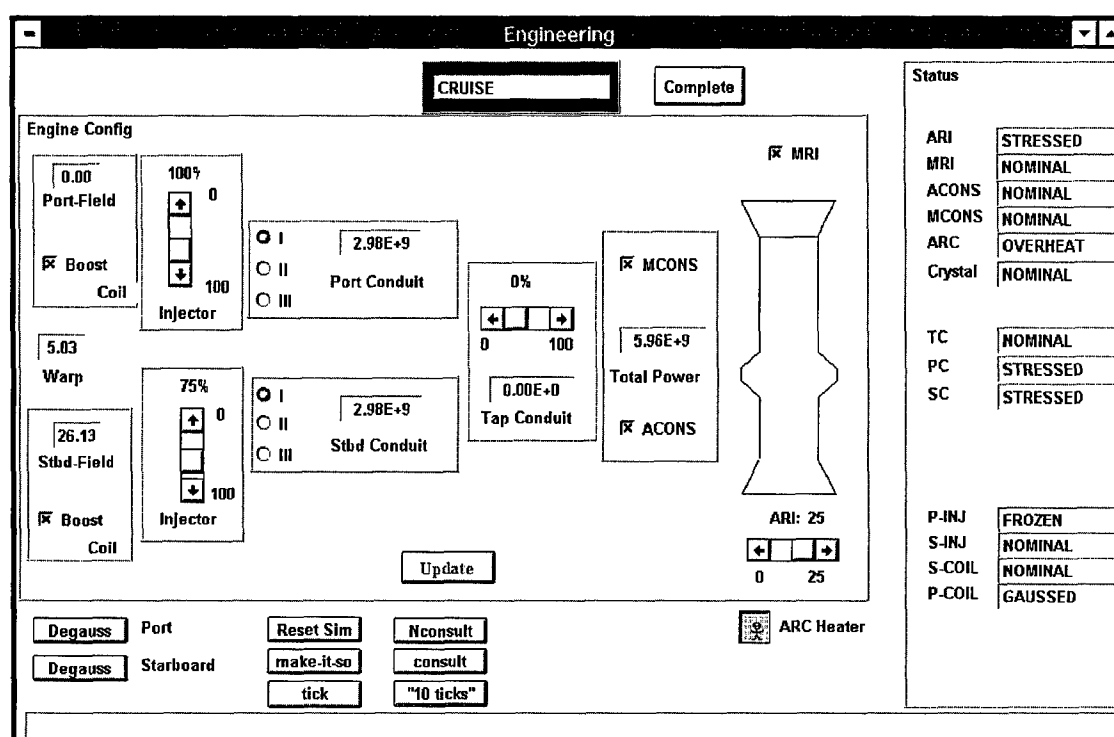


Figure 19. Simulation Interface

Figure 19 shows a typical screenshot of the GUI. The thirteen objects listed on the status display (right side of screen) are the objects that the operator is aware of; the remainder of the objects are hidden in the simulation. The center-bottom bank of six buttons are used for development; the remainder of the controls are used by the operator for

³Symmetrical simulation components are simulation objects that have the same attributes, but represent different instantiations of the object template. (e.g., the left and right engine on a two-engine airplane).

various tasks. The text box at the top of the screen is the directive for reactor configuration, representing the external input to the simulation.

5.2.2.1 Interface. The interface between the simulation and the knowledge base is implemented through a Dynamically Linked Library (DLL). CLIPS is compiled as a DLL, and Lisp accesses the CLIPS functions through a foreign-function call to the CLIPS DLL. This essentially allows Lisp to internally have the functionality of CLIPS. On SCIUS's initialization, the knowledge base is loaded into CLIPS. Knowledge base queries are accomplished by having SCIUS load the system state vector into CLIPS as a set of fact assertions, then run the knowledge base, and finally retrieve any action fact assertions.

5.2.2.2 Simulation Complications. The simulation was designed to be non-trivial, and as such implements a variety of behaviors that differ from a standard action-response model. First, some actions do not cause responses in external objects until after a delay period, while some actions do not cause responses in any objects at all. Some objects enter "problem" states (based upon improper actions), and take an extended period of time to recover after the improper action is remedied, while some objects "break" and do not ever recover. Some objects have "backup" objects that are activated when a primary object is disabled. Some symmetrical objects do not behave identically. Finally, there are many cases where actions on a number of objects control changes of state in other objects. The intent of the simulation complications is to make the simulation "non-tuned" to exploration and induction⁴, thus providing a more rigorous test for the SCIUS approach.

5.2.3 SCIUS Prototype Framework. The bulk of complexity of the SCIUS prototype resides in the Knowledge Acquisition Module (KAM) and the control module. A domain-dependent interface deals with conversions between the simulation state vector and the knowledge base state vector, as well as the return conversion between the knowledge base action facts and the GUI's control inputs. Additionally, the control module is responsible for the scenario exploration process, generating the exploration action inputs to the

⁴To tune the simulation for induction, actions would only affect a single state of a single object, and the results would be immediate. Additionally, objects would never enter "sink node" states, and symmetrical objects would behave identically.

simulation and categorizing the explored scenarios. The KAM contains the ID3 engine, and generates the learned concepts from the explored scenarios to be stored in the expert module. Additionally, the KAM structures the learned concepts into a baseline curriculum to be stored in the tutor module.

5.2.4 Summary. The SCIUS prototype achieves a balance between being complex enough to provide a meaningful research tool, and not being so complex as to involve an extensive amount of development. The SCIUS prototype implements a non-trivial knowledge base and simulation, and the interesting parts of the overall SCIUS architecture. The next section will detail some of the results obtained during the development and testing of the prototype.

5.3 Simulation Issues

Way (53) comments that simulation and GUI development constitutes the majority of the development time for an ICAT system, and my experience with the prototype agrees with this result. Because SCIUS allows for the use of pre-existing simulations, this cost can be eliminated in some cases. Simulation authoring tools can also help minimize the cost of developing interactive simulations. The following sections describe some of the key issues in the design of simulations for the SCIUS architecture.

5.3.1 Interface. There are two key simulation interface issues for SCIUS: the control interface, which allows SCIUS to control the simulation interactively, and the state translation interface, which controls the format for the output of state vector information.

The simulation must be controllable by SCIUS to allow for simulation exploration and demonstration of scenarios. This control can be implemented either through the GUI or through direct object updates.

This process is shown in Figure 20. SCIUS outputs actions, in the form of a control Object-Attribute-Value (OAV) triple. The interface module (which must be written by the developer) converts the action OAV values to either control inputs that happen through the GUI, or to direct object updates to the internal simulation objects. The first method

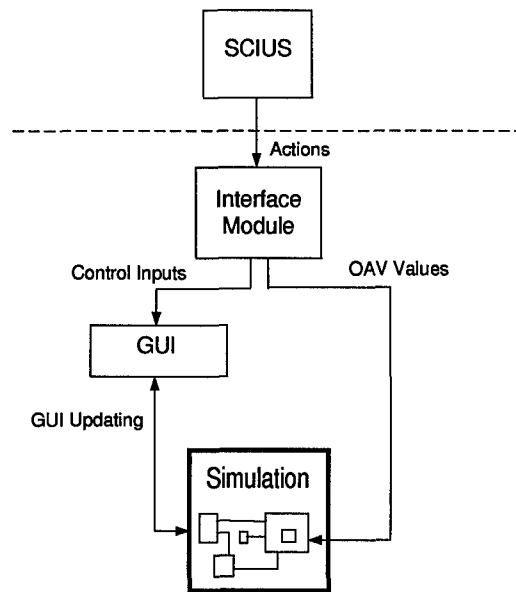


Figure 20. Simulation Control Interface

is much cleaner, because SCIUS controls the simulation through the same interface as the student. This eliminates any negative side-effects (described below) from SCIUS's direct control, and allows an easy method for demonstrating the actions for a particular strategy. This method also allows the simulation to be controlled without knowledge of, or access to, the internal simulation components. The second method (direct object updating) for the control interface requires knowledge of and access to the internal simulation implementation. In this method, the OAV triples are used in code that directly updates the values of internal simulation objects. These changes are reflected in the normal simulation/GUI interface operation. This method may be problematic, however, because functions that are called as a result of GUI inputs will not be called when objects are updated directly.

SCIUS's operation is independent of the method chosen for programmatic simulation control. SCIUS simply passes the action OAV triples to the interface module, which implements the actions appropriately for the simulation. The interface module, however, will have to be customized for a particular domain simulation.

The second aspect of the interface module's operation is the conversion between the simulation's state vector representation, and a representation compatible with SCIUS. SCIUS does not actually parse or otherwise process the raw state vector information, but

it must be able to store it in the scenario library. SCIUS simply stores the state vector in whatever form is required for the simulation, in order to later load it back into the simulation for exploration or training.

5.3.2 Exploration Combinatorics. An important issue in the exploration process is the potential for combinatoric explosion in the number of explored states. The upper bound on the number of states found during state exploration is given by $s \sum_{i=1}^g b^i$, $b \leq B$, where g is the number of explored generations, s is the number of nodes in the original scenario trace, b is the branching factor for each node in a particular exploration session and B is the upper bound on the branching factor, equal to the number of unique elements in the control space. However, it is believed that the number of states discovered in practice will be much lower than this because of two primary factors. First, the theoretical upper bound on the number of states in the state-space is based only upon the number of possible values for each attribute in the state vector. This purely combinatoric measure assumes that all attributes are independent of each other (which gives maximum degrees of freedom). However, attributes without dependencies are not truly part of the simulation. In a real simulation, there will be many dependencies between attributes; these dependencies reduce the potential size of the state space to something typically much less than the theoretical upper bound.

A second limitation is the size of the action vector space. A human-controlled system is not likely to have a huge number of control inputs, and these inputs are not likely to all have a high degree of granularity. In the prototype simulation, there are approximately 100 possible singleton action vectors. Thus, the action vector space is relatively limited. Sampling the action space may lead to duplicate actions. These duplicate actions may cause the same state transition from a variety of different scenarios, limiting the number of possible scenarios found. Additionally, it is often the case that many of the control inputs are binary switches. A random sampling process that finds a switch as a control input has a 50% chance of not changing the value of the switch, resulting in no state transition.

The following discussion presents the results of some exploration experiments run on the prototype simulation. These results are not significant for the population of simulations as a whole, but they do provide some insight into the operation of the exploration process.

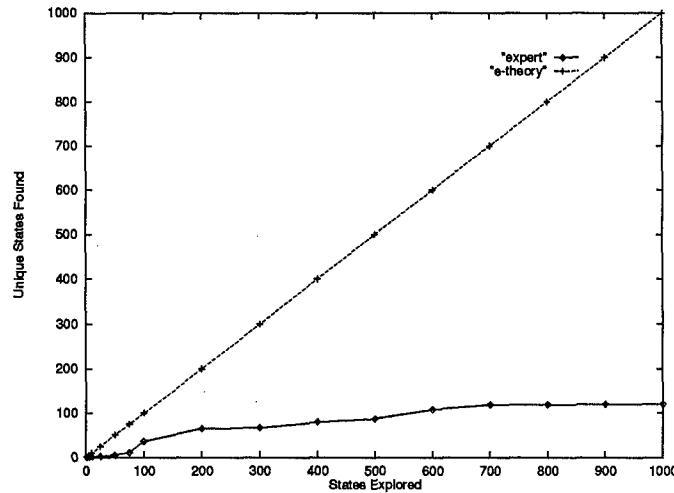


Figure 21. Expert Simulation Exploration

Figure 21 shows a plot of the number of simulation states explored along the expert path against the number of unique simulation states found, along with the theoretical limit of every explored state being unique. It is evident from this graph that the number of unique states found does not increase as quickly as possible, and in fact reaches an asymptotic value at about 120 states in the expert path.

It seems likely that the expert path will always be significantly less than the theoretical exploration limit because the number of correct actions at any given state (one) is so much smaller than the number of possible actions that can be performed, limiting the size of the expert path space.

Figure 22 shows the most simple exploration possible – exploring one generation deep at a branching factor of one, against the theoretical upper bound on number of unique states found. Here, the actual is almost equivalent to the upper bound, because the state space is largely unexplored.

Figure 23 shows the actual versus theoretical number of states found for single generation large branching factor exploration. These graphs show that as the branching factor

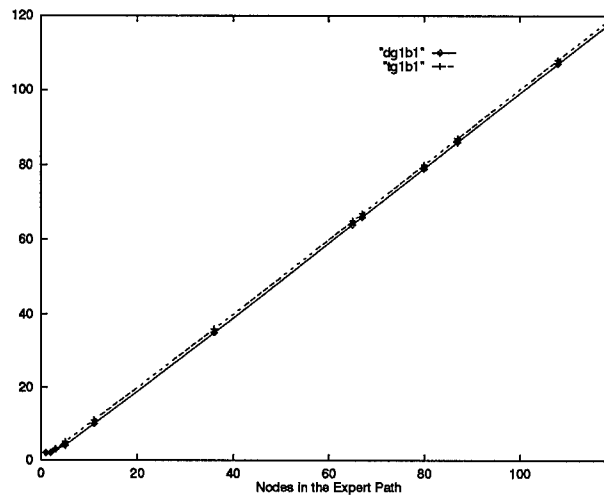


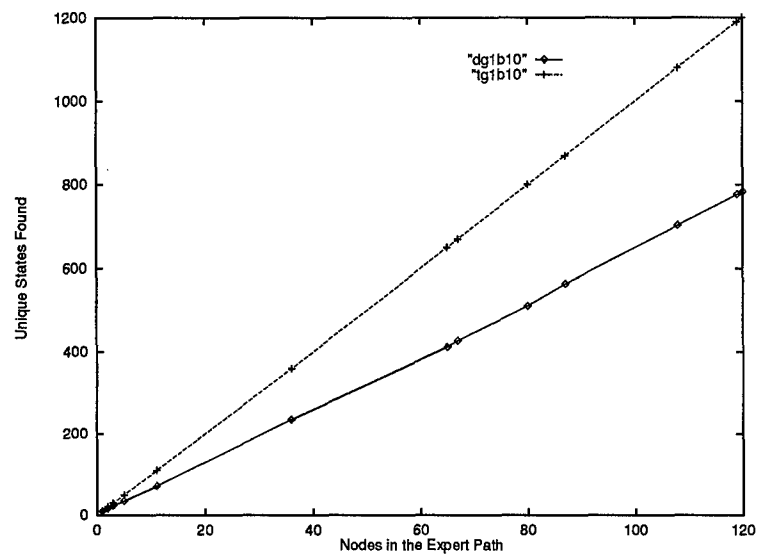
Figure 22. 1 Generation, 1 Branch Exploration

increases, the actual number of unique states found becomes farther removed from the theoretical limit.

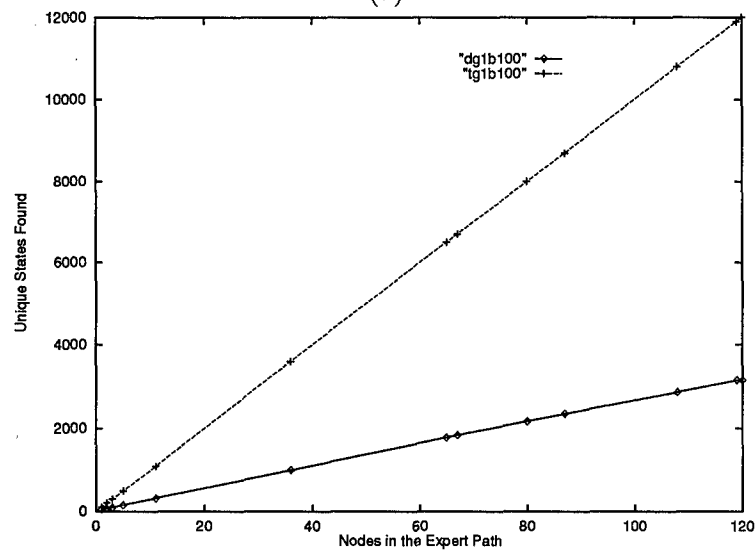
Figure 24 shows the opposite type of exploration, deep with no branching. This type of exploration would not normally be used, because it involves directly searching away from the expert path, which would occur if a student successively performed the wrong action for an extended period of time. These graphs are presented to show that depth versus. breadth does not make a significant difference in the growth of scenarios found during exploration.

Figure 25 shows the number of unique scenarios found in an exploration set-up that is a mix of breadth and depth: two generations deep with a branching factor of five. Again, this type of exploration grows at a much slower rate than the theoretical limit.

The graphs presented demonstrate a number of important issues. First, they show that the prototype simulation is non-trivial; there are at least 2000 unique states reachable in one step from the expert path. Second, they support the supposition that the exploration does not expand at the theoretical combinatoric limit, because of the constraints mentioned above. Although this does not prove that exploration in an arbitrary simulation will be similarly limited, it supports the notion because the prototype simulation is not tuned for ease of exploration.

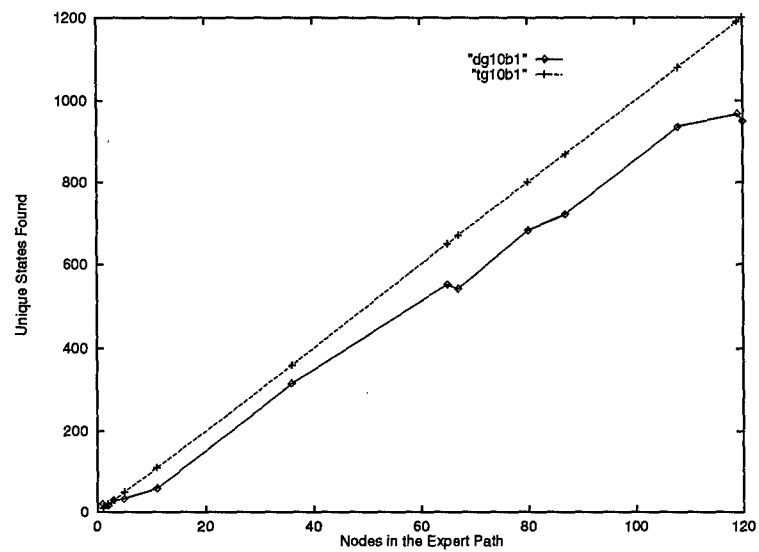


(a)

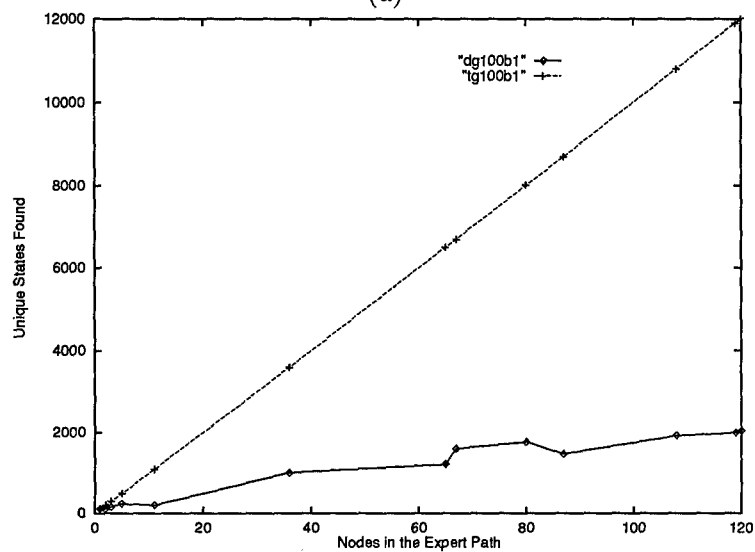


(b)

Figure 23. Simulation Exploration With (a) 1 Generation, 10 Branch, (b) 1 Generation, 100 Branch



(a)



(b)

Figure 24. Simulation Exploration With (a) 10 Generation, 1 Branch, (b) 100 Generation, 1 Branch

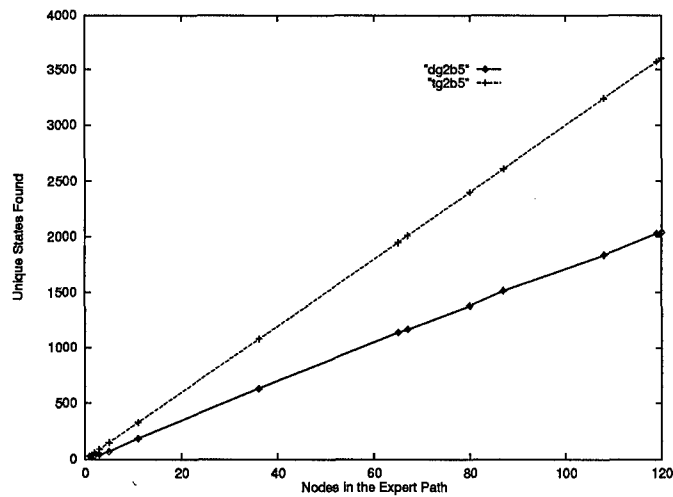


Figure 25. 2 Generation, 5 Branch Exploration

5.4 Knowledge Base Issues

Knowledge base development is fairly straightforward, following a standard knowledge engineering methodology. Of interest to this research is the behavior of knowledge bases in the SCIUS architecture, and the issues involved with interfacing an arbitrary knowledge base to SCIUS.

5.4.1 Interface. The interface module between the knowledge base and SCIUS has two primary functions: converting a simulation state vector to a form compatible with the knowledge base, and converting an action from the knowledge base to an identified OAV triple for use in SCIUS. The first task is fairly straightforward translation, and will have already been accomplished if the simulation and knowledge base are pre-existing components. It is worth noting, however, that the knowledge base form of a simulation state vector (a knowledge base state vector) is frequently going to be much smaller than the simulation state vector. This is because the knowledge base state vector only includes the elements of the simulation that are required for the knowledge base; many simulation objects and attributes are "hidden" from the knowledge base. In the prototype, only 36 of the 206 state vector attributes are required for the knowledge base.

5.4.2 *Timing.* Timing is not addressed in the prototype, because of the efficiency limitations of Lisp and the other pressing research issues. However, timing will be an issue for a real-world implementation of the SCIUS architecture.

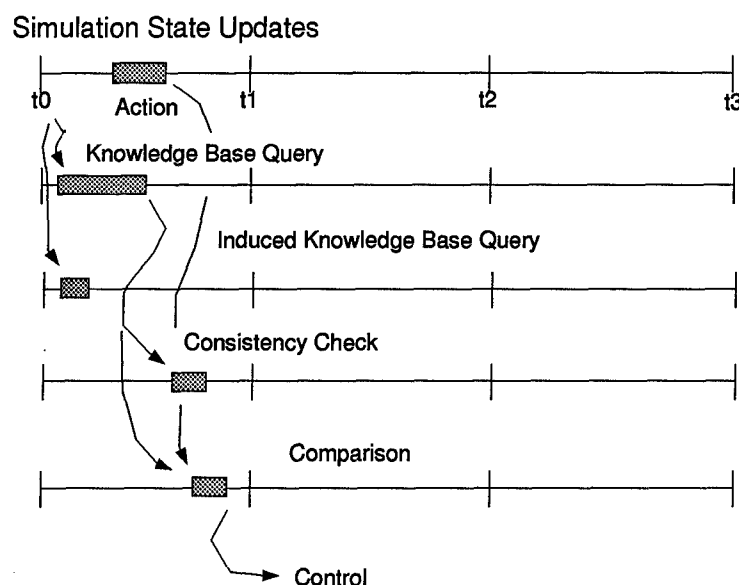


Figure 26. Ideal Timing

Figure 26 shows an ideal timing flow for a typical training cycle. The main trigger is the simulation update (at t_0). This triggers an expert knowledge base query, using the state at t_0 , and a corresponding evaluation of the state using the induced knowledge base. It is likely that the induced knowledge base query will be faster because the induced knowledge base is completely flat, and "running" natively in the language used for SCIUS, whereas the expert knowledge base is external. The result of the knowledge base query is compared to the result of the induced knowledge base query for the purposes of consistency checking. If the expert knowledge base and induced knowledge base are consistent, then the results obtained by the knowledge base must be compared against the action (or inaction) of the student. If the expert action and the student action conflict, then the simulation can be halted for corrective feedback, depending upon the pedagogical strategy. Ideally, this entire sequence occurs during one simulation cycle. However, it could be the case that the action occurs too late in the cycle for the required activities to complete before the next simulation cycle. This should not be a problem because, depending upon the pedagogical

strategy, the simulation will likely be restored to the state at which the student error occurred.

The main timing constraint for SCIUS is that the three activities (knowledge base query, induced knowledge base query, and consistency check) must be accomplished in less than the time required for one simulation cycle. Without this constraint, the simulation would be generating states faster than SCIUS could process them, and there would be no means to clear the backlog of unprocessed states.

5.5 Induction Issues

The measure of success for the induction process is the accuracy and appropriateness of the induced knowledge base. Inductive techniques are typically judged by the number of examples that they can classify correctly in a given set of examples, after being trained with a subset of those examples. The nature of the inductive process makes accuracy and appropriateness a trade-off. If an inductive system is trained with the entire set of examples, then it will develop completely accurate concepts⁵, but they will be specialized to the example set and will not tend to be robust when applied against a new set of examples. On the other hand, if the inductive system is trained on a small subset of the examples, then the system may be more robust, but less accurate. Determining the ideal point for this trade-off is an open issue in the field of machine learning.

It may seem that a training system should be 100% correct and complete. However, in a non-trivial simulation it will be impossible (or at least infeasible) to enumerate all the possible states⁶, so a training system can not be guaranteed to be 100% complete or correct. However, some confidence can be gained that it will be unlikely for a concept to incorrectly classify an example, if the training set is representative.

In the case of SCIUS, the knowledge base is 100% correct within the scope of its design. Ideally, the induced knowledge base would be *completely* equivalent to the original

⁵This assumes there is no noise (incorrect examples), and that the examples are completely classifiable with the features given. This is true for this research, but is not necessarily true for all inductive domains.

⁶It is possible to functionally cover all the possible states in some *educational* domains (such as addition), so a tutoring system could be completely correct and complete in these domains. This is not generally true of training domains.

knowledge base. However, this is an infeasible goal. The only way to *guarantee* that two knowledge bases are equivalent is to enumerate all the possible inputs (infeasible), convert them to simplified algebraic representations (infeasible), or copy the original knowledge base (pointless). At best, we have to settle for some statistical confidence that it is unlikely that the knowledge bases will give different answers. The induction process gives some confidence that the learned concepts are statistically accurate, but the consistency checking process in SCIUS (described in Chapter 4) ensures that an improperly classified scenario will never be presented.

The following sections will present the results from induction experiments conducted on the SCIUS prototype. The purpose of these experiments was to demonstrate the feasibility of induction in this approach and to investigate the behavior of induction in a non-trivial system.

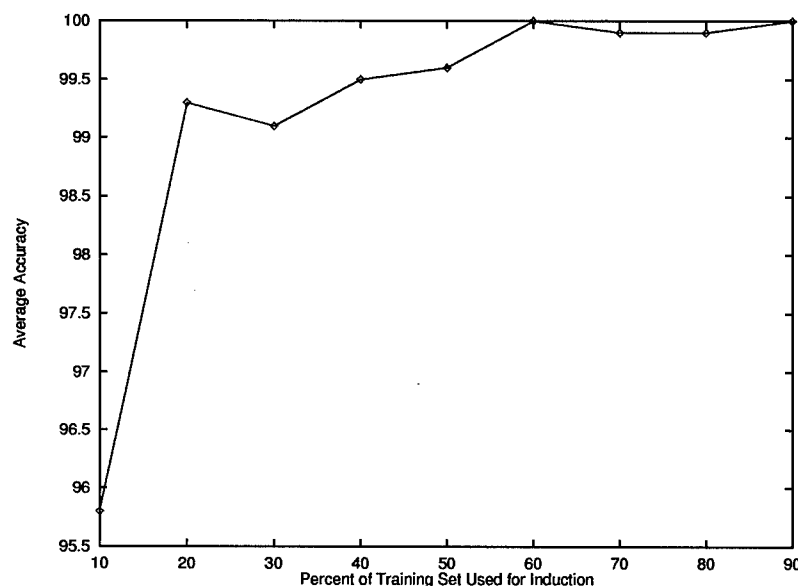


Figure 27. Induction Accuracy

5.5.1 Number of Examples. Figure 27 shows a plot of the accuracy of the induced knowledge base versus the percentage of scenarios used for training. For these tests, a random subset of the example set was used for induction, and the learned concepts were compared for accuracy against the entire set of examples (training with 100% of the example set always results in 100% accuracy). As can be seen in the graph, the accuracy of

the induced concepts quickly rises to a reasonable level as the number of training examples is increased.

This graph indicates that the prototype domain is fairly well-structured; a relatively small subset of the examples is representative of the entire set of examples. Compared to random data, a simulation will impose a large degree of structure, making it more likely that a relatively small representative subset can be found. Additionally, the action-based exploration process finds scenarios that are clustered together in the state space, making it easier to find a small representative sample. A state-based exploration process, on the other hand, will find scenarios scattered around the state space. A random sample of those scenarios will be less likely to be representative of the explored scenarios as a whole. This structure to the scenarios found from action-based exploration supports the idea that a representative set of training examples can be found without requiring an exhaustive search.

5.5.2 Induction Behavior. The prototype knowledge base was designed at the very beginning of this research effort, before most of the issues involving induction and scenario exploration were uncovered. This was done to minimize any conscious or subconscious bias that might have made the knowledge base tuned for induction. It was later discovered that the knowledge base was not only not tuned for induction, it was not a particularly well designed knowledge base either. However, the knowledge base was not improved (corrected), because a sub-optimal knowledge base provides a more realistic and interesting example of a knowledge base that SCIUS might encounter. The implications of the knowledge base problems will be discussed below.

The induced rules have a varying degree of similarity to the original knowledge base rules. The simpler the original rules were, the more likely the induced rules would be exact copies. An example of an exact match was the "degauss" rule:

```
(defrule port-degauss
"Clear the port warp-coil field"
(port-coil state gaussed)
=>
(assert (action port-coil degauss activate))
```

This rule addresses a problem condition that occurs when a coil is brought up to power without activating a coil boost. This type of state is found through exploration; it occurs as the result of an operator error. It is interesting that this rule is matched exactly because this condition can occur any time after the reactor is started. The examples found during exploration are varied enough, however, to allow the induction process to filter out any other potential precondition.

Some induced rules were very close to, but not exact copies of the knowledge base rules. An example of this type of match was the "plasma injector stress" rule. The CLIPS version of this rule was:

```
(defrule port-unfreeze
"Open up a stuck or frozen injector"
(port-injector state frozen | stressed) ;inj is frozen OR stressed
=>
(assert (action port-injector dilation 80))
```

In the simulation, the injectors become stressed if they operate at 100% for a period of time. If the stressed condition is ignored for thirty simulation cycles, then they become frozen. Either of these conditions can be cleared by reducing the injector level. The induced rule was:

```
IF
  (port-injector state stressed)
THEN
  (action port-injector dilation 80)
```

This rule ignores the "frozen" condition because the exploration process never goes deep enough to find a state where the "stressed" condition is ignored for thirty cycles.

Additionally, some oddities were uncovered in the way ID3 handles ranges. An example of this is shown in the following rule:

```
(defrule shut-off-antimatter
"Deactivate antimatter if matter is shut off"
(mri level 0)
(ari level ?a-level)
(test (> ?a-level 0))
=>
(assert (action ari level 0)))
```

This rule simply states that if the matter-reactant injector (MRI) is turned off, then the antimatter-reactant injector (ARI) must also be turned off. The induced rule obtained was:

```
IF
(mri level 0)
(ari level >= 1.0)
THEN
(action ari level 0)
```

This rule is functionally the equivalent of the CLIPS rule, because the ARI can only take integer values. Therefore $(ari > 0)$ and $(ari \geq 1)$ are equivalent expressions. From a pedagogical standpoint, it is likely that $(ari > 0)$ is a better concept to teach, because zero is easier to remember than some arbitrary integer. This example illustrates the blindness of the induction process; to the computer these expressions are *completely* equivalent because there is no knowledge outside mathematical knowledge with which to compare them. It may be the case that rules such as these could be tweaked in a final human expert evaluation of a real-world implementation of SCIUS.

A final example demonstrates a case where an induced rule seems more complex than it needs to be. The following CLIPS rule:

```
(defrule starboard-injector-max
"Configure the injector for max power"
(phase max-warp)
(starboard-injector dilation ?d)
(test (< ?d 100))
=>
(assert (action starboard-injector dilation 100)))
```

This rule is a "configuration" rule that simply states that an injector should be fully open for a maximum power configuration. The induced rule was:

```
IF
(order name max-warp)
(starboard-injector dilation < 95.0)
(pmtc power >= .25)
THEN
(action starboard-injector dilation 100)
```

This rule demonstrates another range oddity, as described above. In the simulation, $(starboard-injector\ dilation < 95.0)$ and $(starboard-injector\ dilation < 100)$ are equivalent

expressions. The unusual aspect of the induced rule is the (pmtc power $\geq .25$) term. This term identifies the port main transfer conduit (PMTc) power as being greater than or equal to .25. This would seem to be an irrelevant term, but in fact it highlights an interesting aspect of the rule induction process.

The prototype knowledge base contains rules that deal with the case where a reactor configuration is called for before the reactor has been started. For example, if the reactor is shut down, and the order is given for cruise configuration, then rules will fire suggesting actions to start the reactor. These "back-up" rules will always fire last, so even though there is a rule with an action setting an injector to 100%, this action will be overwritten by a back-up rule requiring the injectors to be set to 0% (for engine start-up). This combination of rules is functionally correct, but reflects a bad knowledge-base design because facts are overwritten by other rules.

A better design would have been to include an extra precondition in the configuration rules such as (reactor-power > 0). This would ensure that the configuration rules did not fire until the reactor was started. The knowledge base, however, is *functionally* correct, and acts as if it had these preconditions in the configuration rules, because of the interactions between different rules. This functional equivalence is evidenced in the induced rule. The term (pmtc power $\geq .25$) is essentially equivalent to (reactor-power > 0); the terms will always occur together. Induction is blind to the symbolic meaning of the terms; both terms contribute the same amount of information to separating positive and negative examples.

To further explore this phenomena, the induction process was run against a training set of examples that did not involve any back-up rules to start the engine. In this case, the induced rules were virtually identical to the original knowledge base.

Generally, the induced rule base was symbolically very close to the original rule base. Some learned rules matched their original counterparts exactly, while other rules were partially streamlined because of preconditions that were not required in the explored simulation space. Some rules that were outside the simulation space were not learned at all. Finally, some rules were sub-optimal in a pedagogical sense because of the blind choices made by the induction process.

5.5.3 Bias. As previously discussed, the exploration process finds scenarios that are representative of the scenario space that the operator is likely to encounter, and not necessarily representative of the scenario space as a whole. This constraint on the exploration process provides an inductive bias that allows the induction process to discover concepts that are more appropriate to the operator. However, within this constrained scenario space, all scenarios are considered equally likely, because no other information is available to determine likelihood. It may seem as if this is a shortfall, because some scenarios are certainly much more likely than others. However, the operators are typically not being trained to handle *only* the frequently occurring scenarios; the infrequent, anomalous scenarios may be the most important for training. Treating the explored scenarios with equal likelihood increases the chance that an infrequently occurring scenario will be discovered. It also may be the case that the lack of frequency bias will allow the exploration process to discover scenarios that were not anticipated by a human expert. Humans typically have a plethora of biases that allow them to effectively constrain large search spaces. These biases are typically an advantage, but in some cases they blind humans to possible, though unlikely outcomes. This has been the case with some Discovery-Based Learning systems (25, 23), where the systems discovered concepts that the system designer did not anticipate. The exploration and induction process in SCIUS acts as a type of discovery process.

5.6 Limitations of the Model

It is recognized that the SCIUS architecture does not represent an all-encompassing solution for intelligent training. Some training applications will require deeper knowledge training, or optimal performance, and will not be able to benefit from the SCIUS approach. This section will discuss some of the limitation of the SCIUS architecture.

5.6.1 Domain-Specific Knowledge Requirements. As previously discussed, part of the emphasis for this research was to limit the domain-specific knowledge requirements for a training system as much as possible. The following sections discuss the necessary domain-specific knowledge requirements for the SCIUS architecture.

5.6.1.1 Interfaces. An interface must be constructed between SCIUS and the knowledge base, and between SCIUS and the simulation. For the SCIUS to knowledge base interface, knowledge about how to extract the action information from the output of the knowledge base would be required. Additionally, knowledge about how to load the knowledge base with state information (including the conversion from a simulation state vector to knowledge base state vector) and information about how to run the knowledge base would be necessary. For the SCIUS to simulation interface, knowledge about how to control the simulation (loading, starting, stopping) would be required, as well as knowledge about how to extract a simulation state vector from the simulation and convert it to a representation that can be used by SCIUS. Finally, the simulation interface would require knowledge about how to convert an action vector from the knowledge base to an equivalent control input to the simulation.

5.6.1.2 Exploration Knowledge. For exploration to work properly, three specific pieces of knowledge would be required. First, a representation of the action space would be required, so action-based exploration can sample this space. This information should be readily available from the interface development step, however. Second, information about delayed state change nodes (Section 4.4.2.1) would be required so watchdog timers in the exploration process could be set accordingly. Finally, information about any emergency states that are not caused by operator errors would be required, so the exploration process could start at these nodes.

5.6.2 Shallow Knowledge. The most obvious disadvantage of the SCIUS approach (in terms of potential training effectiveness) is the use of shallow knowledge. SCIUS converts a knowledge base (which may already be shallow knowledge) into a completely flat stimulus-response representation. Shallow knowledge is typically not robust; it works within its tightly constrained domain but breaks down quickly in a new situation. It is reasonable to expect operators trained with shallow knowledge to exhibit some of the same behavior.⁷ However, economic constraints are forcing a trend towards using less skilled

⁷There is a significant difference between a human with X amount of knowledge and an expert system with the same knowledge. The human has an entire common-sense background with which to apply that knowledge.

operators with limited education. Shallow knowledge has the advantage that it can be quickly communicated. For example, it takes much less time to train a person to drive an automobile than to train a mechanic. Additionally, this research is not intended to be an all-encompassing training solution. It is designed to provide basic "bootstrap" training, involving bringing a novice to a basic level of competency after which deeper training can be accomplished. One of the most expensive (and likely tedious) aspects of conventional training is one-on-one practice with an expert watching a student. SCIUS addresses this expensive, time-consuming training that must be accomplished before deeper training can begin, if it is required.

5.6.3 Knowledge Base Paradigms. The main intent of this research was to investigate methods for reducing the knowledge engineering requirements for intelligent training systems. This was accomplished by defining an architecture that had minimal knowledge requirements, and was able to use pre-existing knowledge *wherever possible*. The use of pre-existing knowledge is by necessity a secondary priority because of the essentially limitless potential structures of pre-existing knowledge. I attempted to address the most basic form of operator expert knowledge, namely that of a "memoryless" knowledge base that operated only on the current state without any memory of states that have occurred in the past. For example, a power plant operator shuts down a pump because it has just entered a red region on the pressure gauge (memoryless), or he shuts it down because it has been in the yellow region on the pressure gauge for several days (memory). A significant portion of the SCIUS architecture is dependent upon this memoryless assumption; the model essentially breaks down without it. Knowledge base memory would be particularly problematic in any architecture that attempts to use pre-existing knowledge, however.

The main task in using a knowledge base externally is learning why an expert does something at a particular state. In SCIUS, this is accomplished by induction, which works because all the information needed to determine an action is present in one state. However, with knowledge base memory, the triggering states may be an indeterminate number of states in the past. Thus, the information needed to determine the cause of the action is buried somewhere in the stream of states going all the way back to the starting state.

Theoretically, induction could still work in this environment, but it would be horribly inefficient.

A second problematic paradigm for a pre-existing knowledge base is the "automated expert" knowledge base. This type of knowledge base is designed to *act* as a sole operator, and not to support a human operator. This type of knowledge base is less complex than an operator support knowledge base, because it does not need to encompass all the human errors and inefficiencies. If action A is called for at state \vec{S} , then the knowledge base may not need information about what to do when action B is performed at state \vec{S} . The automated-knowledge base may only have knowledge about what to do when everything happens perfectly, so it would not have the knowledge required for adequate induction and scenario exploration for human training.

5.7 Summary

The difficult aspects of the SCIUS architecture were implemented with a non-trivial domain knowledge base and simulation. The performance of the prototype was examined with respect to the growth of simulation exploration, and it was confirmed that the number of unique scenarios found during exploration grows at a much smaller rate than the theoretical maximum. The performance of the inductive process was also examined; induction provided reasonable accuracy when trained on a subset of a large set of examples. The induced rules were generally appropriate and close to the original rules.

There are a number of implementation issues to consider for a real-world implementation of SCIUS. Timing is a key issue because two types of knowledge base consultations (actual and induced) must occur within the span of a single simulation cycle. With respect to the task of interfacing the domain knowledge components to SCIUS, the basic task is simply one of translation. The knowledge base output must be translated to control inputs, and the simulation state information must be translated to a refined representation compatible with the knowledge base.

VI. Conclusions & Recommendations

6.1 Summary

This research has presented an architecture that requires less knowledge for an intelligent training system than other approaches. This is accomplished through the establishment of the minimal knowledge requirements for a training system: a simulation and (at least) shallow knowledge base. The knowledge base provides the basic core knowledge that encompasses the operator's job, but no context. The simulation provides context without knowledge. Together, the interaction of these two components provides knowledge in context, and provides knowledge that is more powerful than either of the components alone.

These knowledge components are used at the interface level only, allowing the internal representation of the components to take any form that meets the interface requirements. This provides enormous flexibility for the implementation of these components, because the plethora of tools available can be used, freeing the system designer from a particular representation. In addition to easing the implementation, this interface-level approach allows the system to take advantage of pre-existing knowledge components, as long as they meet the interface requirements.

A knowledge base and simulation alone are not sufficient for intelligent training. My approach develops the remaining knowledge required for training through two primary tasks: simulation scenario exploration and concept extraction through machine learning induction. Simulation scenario exploration searches the appropriate state space that the operator is likely to encounter, and does not search the entire state space. Concept extraction uses the scenarios found during scenario exploration as a basis for induction. Induction determines the relevant features for a particular concept that distinguish it from other concepts. This is critical information for training the student; without it, the student simply learns stimulus-response behavior – he has no conception of “why” a particular action is performed. The learned concepts are hierarchically organized by complexity and preconditions to provide a baseline curriculum for presentation to the student.

A prototype implementation of the core of the architecture was developed in Common Lisp, as a proof-of-concept. The prototype domain was a fictional antimatter reactor, and involved many of the typical monitoring/controlling activities of a CDS operator. The prototype was non-trivial; the simulation involved state vectors of over 100 attributes and numerous simulation complications, and the knowledge base implemented 55 rules, with a variety of different rule configurations and complexities. The performance of the prototype met expectations. The growth of the simulation exploration was far less than the theoretical upper bound, and the exploration process was able to find a wide variety of scenarios. The induction process was able to discover appropriate rules with a high success rate against unseen examples.

This approach is significantly different from other approaches in the literature. Although authoring systems have been developed, none have been as aggressive with reducing knowledge requirements as my approach. Most authoring systems are simply a template for a particular training approach, requiring a significant knowledge engineering effort from an expert course designer to encode the domain knowledge into the authoring system representation. Additionally, no other research has addressed the issue of using pre-existing knowledge, and very few systems have addressed the use of machine learning in an ITS.

The architecture represents the potential for a significant savings over other approaches for developing intelligent training systems. The architecture allows the opportunistic use of pre-existing knowledge, if available, or minimal requirements for knowledge engineering in comparison to other approaches. The difficult aspects of the architecture have been demonstrated in a non-trivial domain, and performed as expected.

6.2 Objectives

The research objectives presented in Chapter 1 were met by this research effort. The discussion below briefly describes how each of these objectives were satisfied.

Develop an architecture for a training system that uses only a simulation and knowledge base for domain knowledge.

The architecture presented in Chapter 3 uses an automated scenario exploration technique and rule induction (described in Chapter 4) to discover knowledge required for a training model described in Chapter 2. The results from the prototype (described in Chapter 5) support the feasibility of this approach for acquiring domain knowledge from the interface to a simulation and knowledge base.

Develop an approach to automatically generate scenarios for training.

The action-based exploration approach (described in Chapter 4) guarantees legal scenario generation. This approach finds a wide variety of scenarios, clustered around an expert path through the simulation state space. These scenarios will tend to be more representative of the scenarios the operator is likely to encounter than scenarios that are far removed from the expert path.

Develop a technique to isolate the key feature knowledge from a scenario.

Given a representative set of scenarios, the rule induction process (described in Chapter 4) isolates the key features in the scenarios that lead to a particular action. The prototype induction results were promising, with a large number of rules very closely matching their original counterparts. However, some induced rules were correct, but somewhat inappropriate for training, highlighting one of the disadvantages of domain-independent induction.

Determine the knowledge and interface requirements for a simulation and knowledge base in a generic architecture.

The interface to the simulation and knowledge base is described in Chapter 3. Basically, an operator-expert knowledge base inputs a state of the world and outputs a recommended action. A simulation takes an action (set of control inputs), and possibly a simulation control input, then generates a state of the world. This interface is fairly simple, supporting the viability of this interface-level approach. These interfaces require domain specific knowledge, however; this limitation is discussed in Section 5.6.1.

6.3 Contributions

6.3.1 Authoring Systems. SCIUS represents a unique approach to an authoring system, one that can theoretically operate without a course author; the entire domain knowledge can be engineered without knowledge of the fact that it is going to be used for a training system. Additionally, the use of the knowledge components at the interface level represents a unique approach because it allows almost complete freedom for the implementation of these components.

6.3.2 Generic Training System Model. This research presented a model for a generic training system that integrates the necessary domain-specific components of a training system. A model of an interactive time-based simulation was developed and integrated with a model of an operator knowledge base. The interfaces for these components to the generic system were specified and demonstrated in a prototype system.

6.3.3 Automatic Knowledge Acquisition. Two techniques for automatic acquisition of domain-specific training knowledge were developed. A scenario exploration process was developed that explores the scenario space closer to the expert path, and ignores the space far removed from the expert path, while guaranteeing that only legal scenarios will be generated. This should tend to produce states that are more representative of states the operator is likely to encounter. Additionally, the exploration process discovers a wide variety of scenarios that can be used as training and testing examples. In the conventional approach, scenarios have to be developed by hand. In this research, inductive learning was used to discover the important features that lead to actions for particular scenarios. This critical information is required for effective training, because it tells the student "why," not just just "what" to do. A generic baseline curriculum extraction technique was also developed.

6.3.4 Reverse Engineering of Knowledge Bases. As part of the effort to understand the knowledge acquisition process, some interesting aspects of the induction process were discovered that may have application to interface-level reverse-engineering of knowledge bases. If an *appropriate* set of input/output combinations of a knowledge base can be

generated or obtained, it may be possible to reverse-engineer the knowledge base strictly from interface-level behavior. This reverse engineering would obtain a shallow representation of the knowledge base, but that may be quite useful if no other internal representation is available. This may become more important in the near future as older knowledge bases that may be compiled, or run on obsolete shells, become inaccessible. Reverse-engineering may be the only method to recover knowledge that would be otherwise lost. Additionally, the nature of the inductive process causes the induced knowledge base to be partially verified; induction often ignores rules that have syntactic errors because they do not contribute to the knowledge base's behavior. This may have application to the software engineering field as well. Both these approaches depend upon having some means of constraining the possible input space for the knowledge base. In SCIUS, this is obtained by the scenario exploration process, but in some other application the constrained input space could be obtained from historical real-world cases.

6.4 Recommendations for Future Work

The following section discusses some of the possible extensions to this work. This research is a basic foundation effort, intended to explore the feasibility of a new approach. The following tasks would further support the basic research approach, or explore related interesting areas.

6.4.1 Interactive Simulations. The development of an interactive training simulation is probably the most labor-intensive aspect of any intelligent training system. Although non-interactive simulations are well-researched, very little research has been conducted with respect to the unique challenges of building interactive simulations. Interactive simulations basically provide an added level of complexity on top of non-interactive simulations, because issues such as timing, user-interfaces, and psychological fidelity must be addressed.

Research that provided techniques for making the development of interactive simulations easier would have a extremely broad application. Virtually any computer-based training system (SCIUS approach or not) would benefit, as well as many educational do-

mains that use micro-worlds and other types of discovery-based educational approaches. Additionally, psychology, cognitive science and human-factors research efforts could benefit from improvements in interactive simulation construction techniques.

6.4.2 Induction Tuning. For this research, ID3 was not modified, except for a minor change that made ID3 prefer nominal features over numerical features, given that they provide the same information. There are a number of additional changes that could be made to ID3 to make it more appropriate for inducing concepts that will be taught to humans, based upon standard human-factors principles, which would be domain-independent. For example, numerical concepts involving zero could be preferred over concepts involving some non-zero value. Additionally, some more advanced modifications could be investigated, such as the idea that if one feature can be used in a large number of learned concepts, then that feature is preferred over an arbitrary scattering of different features in the learned concepts. Furthermore, it may be the case that multiple nominal features are preferred over a single numerical feature. The information-theoretic measure in ID3 could be modified to encompass this idea as well. Additionally, constructive induction could be used to allow for the construction of new feature that aren't explicitly represented in an action (such as the exclusive OR of two features). This may allow for more compact concepts that are better suited for training. Finally, an alternative approach is to allow for post-induction tuning by a human course designer. This could be as simple as allowing ID3 to present a number of alternative concepts and allowing the course designer to pick the most appropriate.

6.4.3 Real-world Testing. As discussed in Chapter 5, a fictional domain was chosen for the prototype domain to provide expediency and flexibility. An investigation of this approach in a *typical* real world domain could provide additional support as to the appropriateness and generality of this approach. Additionally, an in-depth investigation of the attributes of a wide number of operator training domains would also test the generality of the SCIUS approach.

Additional testing could be accomplished by investigating the efficacy of the training model used in SCIUS in a real-world application. This is really an education research task,

but it would provide valuable support with respect to the appropriateness of the SCIUS training model.

6.4.4 Complete ITS Development. As described in Chapter 5, the prototype implements the knowledge acquisition components of SCIUS, and develops the knowledge required to use a standard training model for presentation. An implementation of a complete training system, covering the span from knowledge acquisition to final training, could provide a valuable insight as to the presentation issues involved in the SCIUS approach. Because of the magnitude of this task, a smaller domain could be used, or pre-existing knowledge components could be used if they were available. Additionally, the complete prototype could be implemented in an efficient language, allowing the timing and system efficiency issues to be more fully explored.

6.5 Overall Conclusions

This research has demonstrated that intelligent training is possible with a smaller requirement for knowledge engineering than other approaches. The opportunistic use of pre-existing knowledge, and minimal requirements for knowledge can potentially represent huge savings in the cost of developing and maintaining a complete training system. The approach used in SCIUS is heavily weighted towards economic savings; this approach is not likely to be desirable for all training domains. However, it may represent the only alternative for cost-constrained course designers.

Appendix A. Definitions

This appendix formally defines the terms used in the body of the dissertation. Note that some of these terms may be slightly different from their standard English definitions.

A.1 Simulation

For the purposes of this research, a simulation is a model of a Complex, Dynamic System (CDS), which a human operator controls.

- **Object:** A simulation object is a representation of some real-world object that is being simulated. Simulation objects may be *external* (visible to the student) or *internal* (not visible to the student.)
- **Attribute:** A simulation attribute is a value representing a single characteristic of one object, such as pressure or temperature.
- **State:** A simulation state is a vector consisting of all the attribute of all the objects (internal and external) in the simulation at a particular time. The terms “state” and “state vector” are interchangeable.
- **Simulation State Space:** The state space of the simulation is the space of all possible state vectors.
- **Control Vector:** A control vector is a set of control inputs to the simulation.
- **Control Space:** The control space is the set of all possible control vectors (including the null control vector).

A.2 Knowledge Base

- **Knowledge Base State Vector (KBSV):** A KBSV is a subset (most likely a proper subset) of a state vector, representing the information required for the knowledge base. Internal object attributes are not included in the KBSV, and some external object attributes may not be included as well.
- **Action:** An action is a recommended control vector for a particular state.

Appendix B. Prototype Algorithms

This appendix presents a description of the algorithms implemented in the SCIUS prototype. The basic techniques for these processes are described in Chapters 4 and 5; the following description represents a specific implementation-level algorithm. Note that the techniques described below are not necessarily the most desirable method for implementing the basic algorithm. Some implementation level decisions were controlled by resource (memory) limitations.

B.1 Scenario Exploration

There are two basic exploration functions in the Lisp code: `explore9`, which implements expert path exploration, and `poke-it`, which implements expert path deviations. Normally, this functionality would be encapsulated into one function, but it was separated in this case for resource and testing purposes. These functions use a CLOS scenario object, which is simply a scenario state vector and an associated action (or nil).

The expert path exploration function takes an input of a “seed” scenario and outputs a list of scenarios found along the expert path. The algorithm is as follows:

1. Save the seed scenario to a file.
2. Load the simulation with the seed scenario.
3. Run CLIPS to find a recommended action. This is accomplished through passing the state vector to CLIPS as a list of facts (through the DLL interface), running CLIPS and getting a returned list of facts that contains action facts.
4. Implement the action through the simulation user interface (takes the scenario to a new state).
5. Run CLIPS to find a recommended action.
6. Save the new state and action to a file (optionally skip duplicates)
7. Loop to step number 4

The result of this process is a list of scenario objects. This process runs for a pre-defined number of steps. The expert path growth test described in Chapter 5 simply consisted of running the process for various numbers of steps (1-2000) and counting the number of unique scenarios found.

The algorithm for exploring away from the expert path works in a similar manner. The input is a list of scenarios and the output is a list of scenarios found by sampling the operator action space (the set of all possible control actions) for a pre-defined depth and branching factor. The algorithm is as follows:

1. Iterate over input scenarios:
2. Iterate over depth:
3. Iterate over branching factor:
4. Implement a random action selected from the action space through the simulation user interface, taking the scenario to a new state.
5. Find the action required for the new state by performing a CLIPS consultation. This is not required for exploration, but will be required for induction.
6. Build a scenario object (state and action) and save it to a file. The object consists of a state and its associated action.
7. Loop through step 3 for the number of the branching factor for a particular scenario.
8. Loop through step 2, performing the branching factor number of branches for each scenario found in the step 6.
9. Loop through the scenarios in the original input set, performing the previous steps.

Basically, this process branches from each original scenario by the branching factor number, then recursively branches from the discovered scenario for the depth. This process is essentially a breadth-first search. The scenario exploration tests described in Chapter 5 involved performing this process for different branching factors and depths, and examining the number of unique scenarios found. One obvious extension to this algorithm would be to explore an additional step determined by the action found in step 5; this would be the

first “recovery” step. This would further enforce the appropriateness of the scenarios found during this process. This was not implemented in the research code, however.

B.2 Induction

This research used ID3 as an induction engine, so only the interface to ID3 was implemented. The interface to ID3 is quite simple; the input is a list of scenarios and the output is a list of concept objects. A concept object is a list of bound (or range-bound) attributes and an associated control input. The ID3 interface algorithm is as follows:

1. Generate a list of all the unique control inputs in all the actions in the training set of scenarios.
2. Iterate over each unique control input:
3. Temporarily label the examples in the training set. If they contain the control input, they are positive, otherwise negative.
4. Translate the scenario state vectors to an ID3 compatible format. One vector is one example.
5. Run ID3, getting a decision tree.
6. Traverse the decision tree paths to positive nodes, extracting the positive example concepts (disjunctive list of conjuncts).
7. Generate a concept object that consists of the induced concept from the previous step and the associated action.
8. Loop through step 2.

This process results in a list of concepts that can be treated as a set of flat rules. The accuracy tests described in Chapter 5 involved using a set of 1000 example scenarios and running ID3 over various subsets of this set. The resulting induced rules were then checked for accuracy over the entire set.

B.3 Code Availability

The Lisp code used in this research is available at

<http://www.afit.af.mil/Schools/EN/ENG/Labs/AI/Research/its.html>.

This code is made available primarily for instructional purposes; it is not robust, demonstration type code. The algorithms described above should be relatively easy to implement in virtually any language. The Lisp code should be treated as one possible implementation. Instructions for using the code are included with the Lisp source.

Bibliography

1. Anderson, John R. *The Architecture of Cognition*. Harvard University Press, Cambridge, MA, 1983.
2. Anderson, John R. "The Expert Module." *Intelligent Tutoring Systems* Lawrence Erlbaum Associates, 1988.
3. Anderson, John R., et al. "Intelligent Tutoring Systems," *Science*, 228:456-462 (1985).
4. Biegel, John E. "An Intelligent Simulation Training System." *Proceedings of the 3rd Annual Workshop on Space Operations, Automation and Robotics (SOAR '89)*, Johnson Space Center, Houston, Texas. 579-584. 1989.
5. Brown, J.S., et al. "DEBUGGY: Diagnosis of errors in basic mathematical skills.." *Intelligent Tutoring Systems* Harcourt Brace Jovanovich, Academic Press, 1982.
6. Brown, J.S., et al. *Intelligent Tutoring Systems*, chapter SOPHIE I, II and III. Harcourt Brace Jovanovich, Academic Press, 1982.
7. Carbonell, J.R. *Mixed-initiative Man-computer Instructional Dialogues*. Technical Report, Bolt Beranak and Newman, 1970.
8. Clancey, William J. *Knowledge-based Tutoring*. The MIT Press, 1987.
9. Clancey, William J. "Tutoring Rules for Guiding a Case Method Dialogue." *Intelligent Tutoring Systems* Lawrence Erlbaum Associates, 1988.
10. Cohen, Daniel I. A. *Introduction to Computer Theory*. John Wiley & Sons, Inc., New York, 1991.
11. Crossman, E.R. "A Theory of the Acquisition of Speed-Skill," *Ergonomics*, 2:153-166 (1959).
12. Derry, Sharon J. and Susanne P. Lajoie. "A Middle Camp for (Un)Intelligent Computing." *Computers as Cognitive Tools* Lawrence Erlbaum Associates, 1993.
13. Elsom-Cook, Mark. "Guided Discovery Tutoring." *Guided Discovery Tutoring - A Framework for ICAI Research 1*, Paul Chapman Publishing Ltd, 1990.
14. Farr, Marshall J. and Joseph Psotka. "A Thematic Introduction." *Intelligent Instruction by Computer: Theory and Practice* edited by Marshall J. Farr and Joseph Psotka, Washington: Taylor & Francis, 1992.
15. Feigenbaum, E.A. "Themes and Case Studies of Knowledge Engineering." *Expert Systems in the Micro-Electronic Age* edited by D. Mitchie, Edinburgh University Press, Edinburgh, Scotland, 1975.
16. Fink, Pamela. "The Role of Domain Knowledge in the Design of an Intelligent Tutoring System." *Intelligent Tutoring Systems - Evolutions in Design* edited by H.L. Burns, et al., Hillsdale, NJ: Erlbaum, 1990.
17. Gagne, R.M. *The Conditions of Learning and the Theory of Instruction*. New York: CBS College Publishing, 1985.

18. Gilbert, T.F. "On the Relevance of Laboratory Investigation of Learning to Self-Instructional Programming." *Teaching machines and programmed instruction* 478, National Education Association, Washington, D.C., 1960.
19. Gonzalez, Avelino J. and Douglas D. Dankel. *The Engineering of Knowledge-Based Systems*. Prentice-Hall:Englewood Cliffs, NJ, 1993.
20. Hergenhahn, B.R. *An Introduction to Theories of Learning*. Englewood Cliffs, NJ: Prentice Hall, 1988.
21. Hollan, J.D., et al. "STEAMER: An Interactive Inspectable Simulation-Based Training System," *AI Magazine*, 5(2):15-27 (1984).
22. Jung, Namho and Taha Sidani. "Tutoring in a Generic Intelligent Simulation Training System," *Computers and Industrial Engineering*, 23(1-4):373-375 (1986).
23. Kilpatrick, F.A. *An Investigation of Discovery-Based Learning in the Route-Planning Domain*. MS thesis, Air Force Institute of Technology, 1992.
24. Klahr, David, et al., editors. *Production System Models of Learning and Development*. MIT Press, Cambridge, Massachusetts, 1987.
25. Lenat, Douglas B. "EURISKO: A Program that Learns New Heuristics and Domain Concepts," *Artificial Intelligence*, 21:61-98 (1983).
26. Lu, Ruqian, et al. "On Automatic Generation of Intelligent Tutoring Systems." *Proceedings of the 1995 Artificial Intelligence in Education Conference*. 67-74. 1995.
27. Miller, M.D. "Applying Component Design Theory to the Design of Coursework." *Instructional Designs for Microcomputer Coursework* edited by D.H. Jonassen, Hillsdale, NJ: Lawrence Erlbaum, 1988.
28. Morrison, John E. *Training for Performance - Principles of Applied Human Learning*. John Wiley & Sons, New York, 1991.
29. Orey, Michael, et al. "Development Efficiency and Effectiveness of Alternative Platforms for Intelligent Tutoring." *Proceedings of the 1993 Artificial Intelligence in Education Conference*. 1993.
30. Papert, Seymour. "Microworlds: Transforming Education." *Artificial Intelligence and Education 1*, Ablex Publishing, 1987.
31. Patrick, John. *Training: Practice & Research*. Academic Press, London, 1992.
32. Quinlan, J.R. "Induction of Decision Trees," *Machine Learning*, 1:81-106 (1986).
33. Rasmussen, Jens. *Information Processing and Human-Machine Interaction*. Elsevier Science Publishing Co., New York, 1986.
34. Regian, J. Wesley. "Representing and Teaching High Performance Tasks Within Intelligent Tutoring Systems." *Intelligent Tutoring Systems* Hillsdale, NJ: Lawrence Erlbaum Associates, Inc, 1991.
35. Reisner, B.V., et al. "Dynamic Student Modelling in an Intelligent Tutor for LISP Programming." *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*. 1985.

36. R.J.DeJong. "The Effects of Increasing Skill on Cycle-Time and Its Consequences for Time Standards," *Ergonomics*, 1:51-60 (1957).
37. Schaefer, B.A, et al. "Knowledge-Based Intelligent Tutoring for Spacecraft Operations." *Proceedings of the Contributed Sessions 1991 Conference on Intelligent Computer Aided Training*. 1991.
38. Schmidt, Richard A. and Robert A. Bjork. "New Conceptualizations of Practice: Common Principles in Three Paradigms Suggest New Concepts for Training," *Psychological Science*, 3-4:207-217 (1992).
39. Shannon, Robert E. *Systems Simulation - The Art and Science*. Prentice-Hall, Inc., NJ, 1975.
40. Shiffrin, R.M. and W. Schneider. "Controlled and Automatic Human Information Processing: II. Perceptual Learning, Automatic Attending and a General Theory," *Psychological Review*, 94:127-190 (1977).
41. Shortliffe, E.H. *Computer-based Medical Consultations: MYCIN*. American Elsevier, 1976.
42. Shute, Valerie J. *Individual Differences In Learning From an Intelligent Discovery World: Smithtown*. Technical Report AFHRL-TP-89-57, Manpower and Personnel Division, Brooks Air Force Base, 1990.
43. Shute, Valerie J. and Josepf Psotka. *Intelligent Tutoring Systems: Past, Present, Future*. Technical Report AL/HR-TP-1994-0005, USAF, Armstrong Laboratory, 1994.
44. Shute, V.J., et al. "Modeling Practice, Performance, and Learning." *Simulation-Based Experiential Learning* edited by D.M. Towne, et al., 133-145, Berlin:Springer-Verlag, 1992.
45. Sime, Julie-Ann and R.R. Leitch. "A Specification Methodology for Intelligent Training Systems," *Computers in Education*, 20:73-80 (1993).
46. Skinner, B.F. *Science and Human Behavior*. New York: Macmillan, 1953.
47. Skinner, B.F. "Teaching Machines," *Science*, 128:969-977 (1958).
48. Sternback, Rick and Michael Okuda. *Star Trek: The Next Generation Technical Manual*. Pocket Books, New York, New York, 1991.
49. Stevens, A., et al. "Misconceptions is Students' Understanding." *Intelligent Tutoring Systems* Lawrence Erlbaum Associates, 1988.
50. Towne, D.M. and A. Munro. "Two approaches to simulation composition for training." *Intelligent Instruction by Computer: Theory and Practice* 105-125, Washington, DC:Taylor and Francis, 1992.
51. Truskowski, Walter F. *Intelligent Tutoring in the Spacecraft Command/Control Environment*. Technical Report N89-19855, NASA, 1989.
52. VanLehn, Kurt. "Student Modeling." *Intelligent Tutoring Systems* Lawrence Erlbaum Associates, 1988.
53. Way, Robert D. "Intelligent Computer-Aided Training Authoring Environment." *Proceedings of rhe Dual-Use Space Technology Transfer Conference*. 1994.

54. Wenger, Etienne. *Artificial Intelligence and Tutoring Systems*. Los Altos, California: Morgan Kaufmann Publishers, Inc, 1987.
55. Woolf, Beverly, et al. "Teaching a Complex Industrial Process." *Artificial Intelligence and Education 1*, Ablex Publishing, 1987.
56. Yazdani, Masoud. "Intelligent Tutoring Systems: An Overview." *Artificial Intelligence and Education 1*, Ablex Publishing, 1987.

Vita

Captain Freeman A. Kilpatrick Jr. was born in [REDACTED] [REDACTED] [REDACTED]. He graduated from Parkway High School, Bossier City, Louisiana in 1983. Following high school, he attended Texas A&M University, College Station, Texas, graduating in 1987 with a Bachelor of Science degree in Electrical Engineering and an Air Force commission. After commissioning, his first assignment was to the Space Surveillance and Tracking System Program Office at Space Systems Division, Los Angeles Air Force Base. During this assignment, he obtained a Master of Science degree in Systems Management from the University of Southern California, graduating in January 1991. He entered the Air Force Institute of Technology in June 1991, obtaining a Master of Science degree in Computer Engineering in 1992.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE March 1996		3. REPORT TYPE AND DATES COVERED Ph.D. Dissertation
4. TITLE AND SUBTITLE A Generic Intelligent Architecture for Computer-Aided Training of Procedural Knowledge			5. FUNDING NUMBERS	
6. AUTHOR(S) Freeman A. Kilpatrick Jr., Capt, USAF				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Department of Electrical and Computer Engineering, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio 45433-7765			8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/DS/ENG/96-02	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) LtCol Nancy Crowley Satellite Control and Simulation Division PL/VTQ Air Force Phillips Laboratory Kirtland Air Force Base, New Mexico 87117			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release, distribution unlimited			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) Intelligent Tutoring System (ITS) development is a knowledge-intensive task, suffering from the same knowledge acquisition bottleneck that plagues most Artificial Intelligence (AI) systems. This research presents an architecture that requires knowledge only in the form of a shallow knowledge base and a simulation to produce a training system. The knowledge base provides the basic procedural knowledge while the simulation provides context. The remainder of the knowledge required for training is learned through the interaction of these components in a state-space scenario exploration process and inductive machine learning. These knowledge components are used only at the interface level, allowing the internal representation to take any form that meets the interface requirements. A prototype of this architecture is implemented as a proof-of-concept to illustrate the viability of the key knowledge acquisition techniques.				
14. SUBJECT TERMS intelligent tutoring systems, machine learning, induction, knowledge acquisition, intelligent training			15. NUMBER OF PAGES 122	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to *stay within the lines* to meet *optical scanning requirements*.

Block 1. Agency Use Only (Leave blank).

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C - Contract	PR - Project
G - Grant	TA - Task
PE - Program Element	WU - Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. (If known)

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

DOD - See DoDD 5230.24, "Distribution Statements on Technical Documents."

DOE - See authorities.

NASA - See Handbook NHB 2200.2.

NTIS - Leave blank.

Block 12b. Distribution Code.

DOD - Leave blank.

DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.

NASA - Leave blank.

NTIS - Leave blank.

Block 13. Abstract. Include a brief (*Maximum 200 words*) factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code (*NTIS only*).

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.